Thèse

pour obtenir le grade de docteur de l'Université de Savoie

Spécialité : Mathématiques

Présentée par Patrick THÉVENON

Titre :

Vers un assistant à la preuve en langue naturelle

Soutenue le 5 Décembre 2006 devant le jury composé de :

Philippe de GROOTE	Rapporteur
Aarne RANTA	Rapporteur
René DAVID	Directeur de Thèse
Laurence DANLOS	Examinatrice
Gérard HUET	Examinateur
Christophe RAFFALLI	Examinateur

Université de Savoie

"Estas verdades não são perfeitas porque são ditas.
E antes de ditas pensadas.
Mas no fundo o que está certo é elas negarem-se a si próprias.
Na negação oposta de afirmarem qualquer cousa.
A única afirmação é ser.
E ser o oposto é o que não queria de mim."

Poemas Inconjuntos
Alberto Caeiro (Fernando Pessoa)

Remerciements

J'aimerais remercier ceux qui ont suivi mon travail de loin et m'ont soutenu : mes parents.

J'aimerais également remercier ceux qui ont suivi mon travail de plus près, et donné des conseils, des idées : R. David, C. Raffalli.

Je remercie également les membres du jury, particulièrement les personnes qui ont accepté d'être rapporteurs sur cette thèse : P. de Groote, A. Ranta et L. Danlos, G. Huet.

Je tiens à remercier les autres personnes qui m'ont cotoyé pendant ces trois ans. La période de la thèse n'a pas toujours été placée sous le règne de l'optimisme, mais la dépression a été évitée grâce à une bonne entente avec mes collègues : Serge, Thomas, Térence, Frédéric et les autres.

D'autres remerciements pour des raisons diverses et variées pour L. Vuillon, P. Verovic, N. Mari, P. Michel.

Merci enfin à tous mes amis et à ceux qui se reconnaitrons dans le terme 'ami'.

J'espère que personne ne se sent oublié. Dans le cas contraire, je remercie les personnes malencontreusement oubliées.

Table des matières

Introduction

La logique et la linguistique sont liées depuis assez longtemps. Jusqu'à présent, c'est principalement la linguistique qui a utilisé la logique comme outil, tout d'abord pour la sémantique, puis pour la syntaxe avec les grammaires catégorielles.
Ici nous allons voir l'utilisation de la linguistique comme outil de la logique. Cette thèse s'est en effet développée dans le cadre du projet DemoNat qui a pour but l'analyse et la vérification de démonstrations mathématiques écrites en langue naturelle.
Ce projet rentre dans le cadre d'une Action Concertée Incitative (ACI), section "Nouvelles interfaces des mathématiques". Il a concerné trois équipes de recherche, comportant des linguistes et des logiciens :
– équipe de logique du LAMA, Université de Savoie ;
– projet Calligramme du LORIA, Nancy ;
– équipe Lattice, TaLaNa, Université Paris VII.

Le premier chapitre sera une introduction à cette thèse. Il aura pour but de présenter plus en détail le but du projet, sans entrer dans les limitations techniques, en donnant une architecture générale du système. Nous décrirons à partir d'exemples les difficultés que nous avons rencontrées lorsque nous avons réfléchi aux problèmes de l'analyse des textes mathématiques écrits en langue naturelle. Nous évoquerons deux outils linguistiques (les ACGs et la SDRT) qui peuvent être utilisés pour l'analyse et servir à la traduction des preuves dans un langage restreint.

Dans le second chapitre nous introduirons un langage restreint, qui a pour but d'être un intermédiaire entre la preuve en langue naturelle et la validation formelle par machine. Il s'agit en fait d'un langage de description de preuves par règles de déduction. Ce langage a pour objectif d'être plus simple que le langage naturel, et d'être interprétable formellement. Il est donc possible, à partir d'une phrase dans ce langage, d'extraire un arbre de règles de déduction, qui devront être validées par la suite. Nous verrons pour les règles complexes comment donner une formule dont la preuve assure leur validité. La syntaxe de ce langage peut faire penser à celle du langage Mizar, dans le sens où il utilise peu de vocabulaire, mais son interprétation est différente car elle fait toujours appel à un démonstrateur automatique qui doit être capable de suivre les indications données.

Le troisième chapitre sera consacré à un point particulier de la thèse, entrant toujours dans le cadre du projet. Nous introduirons en effet un λ-calcul typé faisant partie d'une extension des ACGs. Sur ce λ-calcul, qui comporte deux types de variables et deux types de flèches, nous démontrerons des propriétés théoriques sur le type principal de certains termes. En effet, la présence de deux flèches et l'absence de distinction de l'application implique que l'algorithme de typage nécessite l'introduction de flèches sous-spécifiées. Vouloir faire disparaître ces flèches sous-spécifiées revient à chercher des fragments dans lesquels il est possible de le faire.

Dans le quatrième chapitre nous décrirons le démonstrateur qui a été développé pour ce projet. Celui-ci a pour but de valider les règles décrites par le langage restreint. C'est principalement un démonstrateur par résolution, qui fait une décomposition paresseuse des formules, c'est à dire parallèlement à la preuve. Il y a également l'utilisation de poids qui permettent d'ordonner les formules dans une chaîne de manipulations, qui provient des indices qui peuvent être donnés par le langage restreint. Nous parlerons des stratégies mises en oeuvre pour améliorer

son efficacité, comme l'élimination des subsomptions, la séparation sans séparation et la résolution linéaire.

Le cinquième chapitre a pour but de définir un système de déduction logique, dont le démonstrateur est la base, pour une logique du premier ordre. L'idée est de donner un cadre théorique à ce démonstrateur particulier, et de donner quelques stratégies complètes. En fait, deux systèmes seront donnés, un premier appelé système logique et un second appelé système implémenté. Les preuves de complétude et de stratégies données seront syntaxiques pour le système logique, et sémantiques pour le système implémenté. Ces deux systèmes peuvent faire penser à la fois à la déduction libre de M. Parigot (cf. [Pa]) dans le sens où il s'agit uniquement de règles d'élimination et au calcul des structures de A. Guglielmi (cf. [BruTi]) dans le sens où il n'y a pas de branchement des règles et où l'on voit une conjonction de disjonctions.

Chapitre 1

Présentation générale du projet

1.1 La preuve formelle assistée par ordinateur

1.1.1 Les assistants de démonstration

Il existe de nombreux assistants de démonstration, parmi lesquels nous pouvons citer :

- Alfa (2000, cf. [Alfa]) : Il s'agit d'un successeur de l'éditeur de preuves ALF, qui permet la manipulation d'objets de preuves dans un cadre logique basé sur la théorie des types de Martin-Löf. Il est possible de définir des théories, par les axiomes et les règles de déduction, de formuler des théorèmes et de construire leurs preuves. Toutes les étapes de la preuve sont vérifiées pour empêcher les preuves erronées.
- Automath (1967, cf. [Auto]) : Automath est basé sur un langage créé par N.G. de Bruijn pour représenter les preuves mathématiques à l'aide d'un ordinateur. Il s'agit de la première grande réalisation d'un projet de vérification automatisée pour les mathématiques. Ce programme peut être vu comme le prédécesseur des assistants de preuves basés sur la théorie des types tels que Coq.
- Coq (1985, cf. [Coq]) : C'est un système de manipulation de preuves mathématiques formelles. Les preuves sont mécaniquement vérifiées par la machine. Il est possible d'énoncer des théorèmes, des spécifications de programme, et de développer interactivement leurs preuves à l'aide de tactiques. Le programme est basé sur le calcul des constructions inductives, étendu avec un système de développement modulaire des théories.
- ETPS (1986, cf. [TPS]) : Cet outil est basé sur le démonstrateur TPS (theorem proving system), et est utilisé par les étudiants, afin qu'ils puissent prouver des théorèmes interactivement.
- Isabelle (1991, cf. [Isa]) : Comparé aux autres outils, Isabelle, développé par Paulson et Nipkov [NiPa], se veut flexible. La plupart des assistants de preuve sont construits autour d'un unique calcul formel, en général la logique d'ordre supérieur. Isabelle a la capacité d'accepter une variété de calculs formels. La version distribuée utilise la logique d'ordre supérieur, mais également l'axiomatique de la théorie des ensembles, et plusieurs autres.
- LEGO (1994, cf. [LEGO]) : Un système de développement de preuves interactif, implémenté par Randy Pollack [Po]. Il implémente également plusieurs systèmes de types : le 'Logical Framework' d'Edimbourg, le calcul des constructions, le calcul des constructions généralisé et la théorie unifiée des types dépendants.
  Les preuves sont développées dans le style de la déduction naturelle. La synthèse d'argument et le polymorphisme permettent de rendre la formalisation proche des mathématiques informelles.
- Mizar (1973, cf. [Mizar]) : Le système Mizar est l'implémentation du langage Mizar, dérivé du langage mathématique. Le principe est de faire un langage compréhensible par les mathématiciens et assez rigoureux pour pouvoir vérifier les preuves à l'aide d'un ordinateur. Le projet a pour but pricipal, depuis 1989, de créer une base d'articles, appelés articles Mizar, qui contiennent ainsi des preuves validées formellement. Cette base contient actuellement plus de 40000 théorèmes.

- PhoX (1994, cf. [PhoX]) : Cet assistant de démonstration est basé sur la logique d'ordre supérieur et est extensible. Un des principes de ce programme est d'être le plus convivial possible, et de demander peu de temps d'apprentissage, celui-ci étant utilisé pour l'enseignement auprès des étudiants de mathématiques.

1.1.2 Des outils plus conviviaux

Pendant longtemps peu de ces assistants étaient dotés d'un mode d'utilisation convivial. Dans la plupart des cas les preuves écrites avec un assistant donné demandent une connaissance des commandes particulières à celui-ci.
Depuis quelques années, il y a la volonté de rendre les assistants plus conviviaux, permettant de simplifier au maximum la tâche des utilisateurs.

- Sur le système Alfa, l'utilisation de GF (Grammatical Framework, 2002) de Aarne Ranta (cf. [Ra3]) permet de traduire les énoncés ainsi que les preuves formelles en un texte en langue naturelle. En effet GF est un langage de programmation pour écrire des grammaires. Il a pour but en particulier de faire des grammaires multi-langues, au dessus desquelles on peut construire des applications comme des traductions et des interactions homme-machine. L'utilisateur de Alfa peut alors relire ses preuves dans un langage naturel, donc plus agréable.
- Dans l'outil d'aide à la preuve Coq (1985), des outils sont créés pour rendre plus agréable la tâche du mathématicien.
  - le 'proof-by-pointing' (cf. [BeKaThe]) est un outil permettant de faire sa preuve en sélectionnant des expressions dans les hypothèses ou le but courants, et d'oublier le langage de preuve. En fonction des sélections, des commandes sont proposées. Celles-ci doivent être applicables et font donc avancer la preuve.
  - WikiCoqWeb est une interface expérimentale développée par Loïc Pottier. L'idée est de trouver le compromis entre les preuves traditionnelles des mathématiciens, et les preuves formelles écrites à partir de Coq. Cette interface permet en particulier de définir des termes qui peuvent être surchargés (par exemple la fonction somme peut être utilisée pour les entiers et pour les réels). Il y a alors un algorithme de résolution qui cherche le bon élément à donner lors de la traduction dans Coq.
  - MMode (cf. [GiWi]) est un mode Mizar pour Coq, pour sa version 7.4. L'idée est d'obtenir un langage de preuve déclaratif, comme en possèdent Automath, Alfa et Mizar, par opposition aux langages de preuve procéduraux que l'on trouve dans Coq et Isabelle par exemple. La différence entre les deux est grossièrement que dans le mode procédural, la preuve part du but et remonte, tandis dans le mode déclaratif la preuve avance des hypothèses vers le but. La frontière n'est pas cependant pas très bien établie entre les deux modes.
- Le système ETPS permet de réordonner les preuves, effacer des parties de preuve, n'afficher que les parties actives des preuves, ainsi que sauver des preuves incomplètes. Un éditeur de formules permet aux étudiants d'extraire des formules nécessaires qui apparaissent en un point de la preuve, et d'en construire de nouvelles à partir d'elles.
- Une extension de Isabelle, appelée Isar (cf. [We1]), est un langage de preuve qui permet de structurer les preuves, et d'obtenir des textes de preuves compréhensibles par l'humain (ainsi que par la machine). Le nom Isar indique que cette extension a en fait été inspirée par Mizar. De manière interne, Isar fonctionne comme une machine à états avec des transitions qui analysent incrémentalement le langage de preuve. Il y a deux modes, un qui autorise le raisonnement en avant, permettant à l'utilisateur d'exprimer des faits, un qui autorise le raisonnement en arrière, en permettant d'utiliser les tactiques usuelles de Isabelle.
- Dans le cas de Mizar, le concept initial est déjà d'utiliser un langage compréhensible y compris par des mathématiciens non utilisateurs.
- Sur l'assistant PhoX ont été développés plusieurs systèmes :
  - un module appelé menu contextuel a pour but de permettre aux utilisateurs de faire leurs preuves presque exclusivement à l'aide de la souris, en cliquant sur les hypothèses à développer, où à utiliser avec d'autres hypothèses. En ce sens il est très proche du système

'proof-by-pointing' de Coq. Ensuite un texte en langue naturelle est formé, donnant ainsi une preuve en français.
- il a aussi été développé new_command, dont le but est de permettre d'écrire des preuves dans un langage très restreint mais proche de la langue naturelle. On se rapproche ici de l'idée du langage Mizar, mais la validation de ces commandes est différente car elles dépendent pour la plupart d'un démonstrateur automatique.

1.1.3 Des langages de preuve plus naturels

Nous avons vu dans la section précédente que certains des outils conviviaux sont basés sur l'utilisation d'un langage proche de celui de la langue naturelle.
Le plus célèbre d'entre eux, le langage Mizar, a inspiré entre autres Isar et MMode. Comme nous l'avons signalé pour MMode, ces langages ont pour but non seulement d'être lisibles par tout mathématicien, mais également d'être des modes de preuve déclaratifs. Ainsi les preuves partent des hypothèses pour aboutir au but.
Une autre façon de voir est que dans le mode procédural on s'attache en général plus à indiquer ce qui est nécessaire pour construire la preuve plutôt que comment l'obtenir. En effet, les hypothèses ajoutées dans les étapes de preuve sont données explicitement, et ainsi on justifie les étapes de preuves par une liste de prémisses, et non par une suite de règles de déductions. Un tel mode est plus proche des preuves en langue naturelle que le mode procédural.
Il demande cependant l'usage d'un démonstrateur automatique pour justifier les étapes de démonstration :
- Dans le cas de Mizar, c'est un programme appelé *justifier* qui se charge de cela. Il a pour caractéristique de préférer la rapidité à la puissance, c'est à dire que les démonstrations doivent passer immédiatement (ou presque), ou bien échouer, ce qui oblige l'utilisateur à éventuellement détailler plus sa preuve.
- Dans le cas de Isar, la justification est plus puissante que pour Mizar, et utilise plus de tactique. Alors que Mizar n'a que deux tactiques, *by* et *from*, Isar en utilise plus : *rule*, *simpl*, *blast* (raisonnement sur le calcul des prédicats), *auto* et *arith*. Celles-ci sont en général plus puissantes que celles de Mizar.
- Dans le cas de MMode, il s'agit d'utiliser des versions modifiées des tactiques de preuve automatique de Coq comme *auto*

Il existe d'autres systèmes de ce style :
- DECLARE (cf. [Sy]) est un langage déclaratif basé sur trois principes :
  – décomposition du premier ordre et enrichissement ;
  –
  – appel à l'automatisation.
  Le vérificateur est à l'image de celui de Mizar très rapide. Chaque but à prouvé est traité : simplifié, séparé en plusieurs cas qui sont prouvés par un démonstrateur utilisant la méthode des tableaux.
- SPL (cf. [Za]) est un langage déclaratif de preuve simple pour le système HOL (cf. [HOL]), également basé sur le langage Mizar. Cependant celui-ci est en plus extensible. L'utilisateur peut en effet étendre la syntaxe et la sémantique du langage pendant le développement d'une théorie. Cela permet alors de réduire les différences entre les preuves formelles et les preuves informelles, et donc d'améliorer la lisibilité. Les preuves écrites dans ce langage sont vérifiées grâce à un démonstrateur utilisant la méthode des tableaux au premier ordre avec égalité. Cette méthode de preuve est alors considérée comme une règle dérivée de HOL.
- SAD (cf. [LyPaVe]), un système Ukrainien (2006), fonctionne sur trois niveaux de raisonnement :

  1. La présentation du texte, où la preuve est écrite dans un langage restreint formel ;
  2. Le raisonnement de haut niveau, où un problème pour démonstrateur automatique est simplifié en plusieurs sous-problèmes plus simples ;

3. La déduction de bas niveau, où l'on fait la preuve des sous-problèmes, dans un démonstrateur de la logique du premier ordre.

Ce système est sensiblement comparable au notre, qui comporte également les trois étapes. Les différences essentielles sont le fait que dans SAD l'utilisateur peut agir le raisonnement de haut niveau, ce qui n'est pas le cas ici. Mais le démonstrateur, dans notre cas, n'est pas nécessairement limité au premier ordre. D'autre part, la méthode utilisée pour le démonstrateur est différente, puisque dans SAD il s'agit de la méthode des tableaux.

1.1.4 Vers des preuves en langage naturel

Le projet DemoNat s'attache tout particulièrement à développer un mode de preuve utilisant un langage compréhensible, tout comme ceux que nous avons décrit dans la section précédente, en allant même plus loin : analyser et valider des preuves écrites en langue naturelle, donc sans restriction de langage.

L'idéal en effet pour le mathématicien souhaitant certifier ses preuves serait de pouvoir écrire celles-ci naturellement, comme il le fait sur le papier, dans un assistant de démonstration qui serait capable de comprendre ses étapes de démonstration et de les valider.

Notons que des tentatives en ce sens on déjà été tentées.

- Vers les années 1960, Paul Abrahams a cherché à écrire un programme en Lisp permettant de vérifier les preuves mathématiques provenant de livres. Il a cependant assez vite du renoncer à ce projet, les difficultés étant très importantes en comparaison des connaissances en programmation pour pouvoir effectuer l'analyse sémantique nécessaire. Il a alors défini un langage formel et un ensemble restreint de commandes de construction de preuve, et implémenté Proofchecker (cf. [Ab]), qui avait pour but de vérifier si le texte d'entrée vérifiait les contraintes formelles.
- Un autre système, Nthchecker (cf. [Si]), dédié aux preuves de théorie des nombres et créé par Simon a pour but d'analyser des preuves informelles de théorie de nombres, et de les vérifier. Mais ce système n'a adopté ni une théorie linguistique, ni une théorie pour modéliser le type de raisonnement que l'on peut trouver dans les textes mathématiques. Ce système reste par conséquent assez limité.
- Claus Zinn (cf. [Zi]) a cherché à étudier par le biais de la DRT les preuves mathématiques trouvées dans des livres, en donnant une structure pouvant être plus tard vérifiée par un système de démonstration. La DRT, dont nous reparlerons brièvement dans 1.4.1, nécessite une extension afin de pouvoir obtenir une adéquation avec le discours mathématique. Claus Zinn propose les structures de représentation des preuves (PRSs pour Proof Representation Structures), qui permettent de rendre compte de la structure du discours mathématiques. Ici l'extension choisie est la SDRT, comme nous le verrons dans 1.4.1.

La suite du chapitre

Ce chapitre a pour but de présenter le projet. Nous décrirons ce qui doit être analysé, quel est le résultat de cette analyse et enfin comment doit être traité ce résultat pour que la preuve soit validée. L'idée est en fait de traduire le texte en langue naturelle vers un langage restreint, qui est donc un langage de preuve déclaratif.

Le texte traduit doit alors être justifié, et pour cela un démonstrateur automatique doit être utilisé. Il n'y a pas dans ce système plusieurs tactiques, mais une seule, qui est l'appel à un démonstrateur qui a pour tâche de suivre au mieux le raisonnement de l'utilisateur.

La description que nous allons donner dans ce chapitre est celle d'un logiciel idéal, qui n'est à ce jour pas implémenté. Par idéal nous entendons ne pas mettre de limitation technique à notre approche, celles-ci apparaissant dans les chapitres suivants. Ce chapitre a donc plus pour but de motiver la suite que de décrire effectivement ce qui a été fait durant ces trois ans par les différentes équipes qui ont participé au projet.

1.2 Une preuve en langue naturelle ...

1.2.1 Le cadre

Décrivons un idéal pour le mathématicien qui souhaite vérifier sa preuve formellement à l'aide d'un ordinateur.
Au tout début, on se place dans un certain cadre, c'est à dire que l'utilisateur donne les pré-requis pour ce qui va suivre. Ce peut être sous la forme d'indications de connaissances déjà aquises et il s'agit alors d'une importation de théories pour la machine. Ce peut aussi être la définition de notions et la donnée d'une axiomatique nouvelle.
Après avoir donné les bases sur lesquelles le travail sera fait, l'utilisateur donne et démontre ensuite les énoncés des différentes propositions. Il doit pouvoir faire cela de manière tout à fait naturelle, comme il le ferait sur une feuille de papier, ou dans un cours.
Le style de chacun étant particulier, il est préférable de ne pas avoir de restriction sur le vocabulaire utilisé ou la forme de la preuve. Cela signifie que l'utilisateur ne doit pas avoir à faire attention à son style, mais juste à la correction de sa preuve. Devoir faire attention à écrire une preuve dans un langage qui n'est pas le sien peut entrainer en effet une perte du cheminement naturel au profit d'une mécanisation, ce que l'on cherche précisément à éviter.
Il faut cependant pouvoir indiquer qu'à certains points de sa preuve, la machine peut analyser ce qui a été fait jusqu'alors et donner l'ensemble des hypothèses et le but à prouver à ce point précis de la preuve, s'il n'y a pas d'erreur jusque là. C'est en cela que c'est un assistant de démonstration :
- le programme redonne les éléments qui sont accessibles à une étape donnée de la preuve ;
- ce qu'il reste à faire pour terminer la preuve est indiqué également ;
- ce qui est fait avant est vérifié et tout passage erroné ou posant une difficulté doit être indiqué par la machine.

La machine peut éventuellement aider à structurer la preuve de l'utilisateur. Si par exemple l'utilisateur a annoncé faire une preuve par cas, la machine peut faire des indentations et préparer les différents cas, que l'utilisateur peut prouver dans l'ordre qu'il souhaite.

1.2.2 Des exemples

Nous allons à partir d'exemples pris sur des cours ou des copies d'étudiants voir quelques preuves, ainsi que des définitions. Nous étudierons celles-ci dans la section 1.3 de ce chapitre. Nous supposons connues les notions de topologie générale et de suites. Dans le cas contraire le lecteur pourra se référer à tout livre de cours de licence de mathématiques.

Donnons deux définitions :

Définition 1.2.1 (continuité en un point) *Soit 2 espaces topologiques X_1 et X_2. Soit f une fonction de X_1 vers X_2 et soit $x \in X_1$. On dit que f est continue en x si, pour tout W voisinage de $f(x)$, $f^{-1}(W)$ est un voisinage de x.*

Définition 1.2.2 (continuité) *Soit X_1 et X_2 des espaces topologiques. Soit f une fonction de X_1 vers X_2. On dit que f est continue si pour tout O ouvert de X_2, $f^{-1}(O)$ est un ouvert de X_1.*

Voici une preuve telle que l'on peut en voir dans un cours d'analyse :

Proposition 1.2.3 *Soit X_1 et X_2 des espaces topologiques. Soit f une fonction de X_1 vers X_2. Si f est continue alors, pour tout x dans X_1, f est continue en x.*

Preuve : Supposons f continue et soit x dans X_1. Posons $y = f(x)$. Soit V un voisinage de y, prouvons que son image inverse est un voisinage de x. Par définition du voisinage, soit O un ouvert inclus dans V et contenant y. Comme f est continue, $f^{-1}(O)$ est ouvert et x appartient à $f^{-1}(O)$. Comme $f^{-1}(O) \subset f^{-1}(V)$, la preuve est terminée. □

Voici une preuve écrite par un étudiant (modèle) d'une proposition sur les suites :

Proposition 1.2.4 *Pour toutes suites $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ et pour tout réel l, si $(v_n)_{n\in\mathbb{N}}$ est extraite de $(u_n)_{n\in\mathbb{N}}$ et si $(u_n)_{n\in N}$ converge vers l, alors $(v_n)_{n\in\mathbb{N}}$ converge vers l.*

Preuve : Soit v_n une suite extraite de u_n avec u_n qui converge vers l, montrons que v_n converge vers l. C'est à dire : soit $e > 0$, on cherche à prouver qu'il existe n_1 tel que pour tout $n > n_1$ $d(v_n, l) < e$. Comme v_n est une suite extraite alors il existe une fonction f croissante telle que $v_n = u_{f(n)}$ pour tout n.
Comme $e > 0$ et u_n converge vers l, alors il existe n_0 tel que pour tout $n > n0$ $d(u_n, l) < e$. Soit $n > n_0$, montrons qu'alors $d(v_n, l) < e$. Mais maintenant, si $f(n) > n_0$, comme $v_n = u_{f(n)}$, alors on aura démontré que v_n converge vers l. Comme f est croissante et que $n > n_0$, on a $f(n) > f(n_0) \geq n_0$, d'où $f(n) > n_0$. □

Le lecteur pourra voir en annexe IV une traduction des preuves de ces propositions dans le langage que nous définirons au chapitre 2.

1.3 ... analysée par machine...

1.3.1 Le cadre

Tout le travail de l'analyse des définitions, des énoncés ou des preuves consiste à comprendre ce qu'a écrit l'utilisateur. Il s'agit d'une véritable analyse sémantique du texte mathématique comportant certes une rigueur dans le raisonnement logique mais également beaucoup d'éléments implicites qui doivent être détectés le plus possible.
Il est en effet observé que les preuves écrites formellement sont beaucoup plus longues que les autres du fait que les raisonnements qui nous paraissent évidents ne sont naturellement pas exprimés.
On peut toutefois penser que cette analyse est plus simple que celle d'un texte littéraire quelconque grâce au fait que la variété des phrases possibles dans les preuves mathématiques semble beaucoup plus faible que dans l'ensemble des corpus littéraires.

1.3.2 Les difficultés de l'analyse

Notre but ici est de montrer quelles peuvent être les difficultés pour l'analyse des preuves écrites en langue naturelle.
Nous n'avons pas pour ambition ici d'apporter des réponses, mais seulement de pointer du doigt certains des problèmes rencontrés même lors de l'étude de preuves considérées comme raisonnables.

Le vocabulaire

La question du vocabulaire est assez importante. Il y a en effet dans la plupart des cas une différence entre l'usage que l'on fait d'une notion dans une preuve en langage naturel et dans une preuve formelle. Il est fréquent qu'une notion puisse être désignée naturellement de plusieurs manières, alors qu'elle doit l'être d'une unique manière dans une preuve formelle.
Donnons ici quelques exemples :

- On souhaite parfois pouvoir utiliser un symbole mathématique et parfois un terme du langage naturel. Par exemple dans la proposition 1.2.3 on a utilisé "soit O un ouvert inclus dans V", et "$f^{-1}(O) \subset f^{-1}(V)$". La notion d'inclusion est ici utilisée de deux manières, posant le problème de la description des formules à l'aide de la langue naturelle. Ce phénomène est tout à fait spécifique à l'analyse de démonstrations mathématiques.
- Il est aussi possible d'utiliser plusieurs formes lexicales d'une même notion. Par exemple la continuité peut être utilisée de plusieurs manières : "continu" étant un adjectif, il peut être accordé en genre (masculin/féminin) ou en nombre (singulier/pluriel). De plus on peut

écrire la phrase "comme f est continue", mais également "par continuité de f", qui désigne la même chose, mais sous une autre forme. Ce phénomène se retrouve de manière courante en traitement automatique de la langue (TAL), il se retrouve naturellement ici.
– Dans certains cas un même mot peut désigner deux notions. C'est le cas lorsque l'on définit quelque chose de deux manières différentes qui s'avèrent équivalentes par la suite. Par exemple l'adhérence d'un ensemble A peut être vue comme l'ensemble des points adhérents à A ou comme l'intersection de tous les fermés contenant A. Ainsi lorsque l'on parle de l'adhérence d'un ensemble, on peut désigner soit une notion soit l'autre, selon ce qui est le plus simple pour le raisonnement. Ce phénomène est également assez courant en linguistique. Il peut cependant paraître surprenant en mathématiques. Dans ce cadre particulier, le référent est le même pour deux définitions différentes, mais il doit y avoir un théorème qui démontre l'équivalence des définitions.

L'implicite

Il y a dans les preuves que l'on écrit de manière naturelle beaucoup d'implicite. C'est ce qui fait par ailleurs qu'une preuve formelle a une taille beaucoup plus importante qu'une preuve en langue naturelle. L'implicite est parfois dû au fait que l'on se trouve dans un contexte particulier, qui fait que lorsque l'on lit une preuve certains détails ne nous sont pas nécessaires pour comprendre.
Il y a bien entendu de l'implicite dans le langage ordinaire, mais dans le cadre de notre application, c'est ce qui rend l'analyse particulièrement difficile, puisqu'une preuve doit être la plus rigoureuse possible, et l'implicite doit absolument être détecté pour qu'il puisse y avoir validation de la preuve.

- Dans la proposition 1.2.3 on voit dans la preuve que l'on parle d'ouverts, sans jamais préciser que ce sont des ouverts de X_1 ou de X_2. Pourtant il est nécessaire du point de vue formel de préciser ce fait (voir la définition 1.2.2). A la lecture de la preuve on n'est pas dérangé car on sait de manière implicite dans quel ensemble sont les éléments, l'information que x est dans X_1 étant suffisante pour comprendre.
- Toujours dans la proposition 1.2.3, relisons un passage de la preuve : "Soit V un voisinage de y, prouvons que son image inverse est un voisinage de x". Il y a dans cette phrase deux choses implicites :
  – La première est que l'on utilise une anaphore, c'est à dire un mot permettant de désigner un objet précédent sans faire de répétition. Ici "son" permet de désigner le voisinage V de y. La résolution des anaphores est l'une des difficultés de l'analyse de textes.
  – L'autre implicite de la phrase est que l'on parle d'image inverse sans préciser de fonction. Ici c'est le contexte qui nous permet de conclure que la fonction est f, puisque c'est la seule qui est utilisée dans cette preuve.
- Un autre fait particulier aux preuves est que le but courant est la plupart du temps implicite. Plus précisément, on n'explicite pas nécessairement la formule que l'on est en train de prouver. C'est en cela que la preuve écrite par l'étudiant de la proposition 1.2.4 est modèle : à chaque fois est indiqué ce qu'il suffit de prouver pour terminer la preuve. Par exemple dans la phrase "Soit v_n [...] montrons que v_n converge vers l".
  Il serait tout à fait possible de comprendre la preuve sans donner la dernière partie de la phrase. C'est ce qui se passe dans la proposition 1.2.3. Cependant pour une preuve formelle, c'est tout à fait insuffisant. L'analyseur a alors la difficile tâche de comprendre que le but a changé, a moins que ce ne soit l'interprétation du langage intermédiaire qui détecte cela, comme nous verrons qu'il est possible de le penser dans le chapitre 2 à la section 2.3.3.
- La preuve de l'étudiant est modèle, pourtant elle donne une autre difficulté à l'analyse, dans la phrase "Mais maintenant, si $f(n) > n_0$, comme $v_n = u_{f(n)}$, alors on aura démontré que v_n converge vers l". Le mot 'si' dans cette phrase peut être difficile à interpréter : on aurait pu penser qu'il signale une preuve par cas ou par l'absurde comme cela peut être le cas assez souvent, or dans la preuve de l'étudiant il s'agit de l'annonce d'un résultat ("$f(n) > n_0$") que l'on va prouver par la suite. De plus, avec l'aide de ce résultat, la preuve

du fait que "v_n converge vers l" est terminée grâce au fait que "$v_n = u_{f(n)}$".
La difficulté est de trouver la bonne interprétation de la phrase, qui n'est pas facile hors de son contexte. C'est en effet après avoir lu les phrases précédentes que l'on sait qu'au moment de cette phrase on cherche à prouver que "v_n converge vers l".

- Une autre difficulté de cette preuve est l'introduction d'éléments nouveaux de manière implicite. En effet, dans la toute première phrase, l'étudiant introduit explicitement v_n mais pas u_n ni l. Ceci est une difficulté, puisqu'il faut savoir si un élément est déjà connu ou s'il est introduit.
  Avec le mot clé "soit" on introduit généralement de nouveaux éléments, ce qui peut aider l'analyse. Dans cet exemple il n'y a pas de mot clé introduisant u_n et l. Une erreur standard lors de preuves formelles assistées par ordinateur est le référencement à des éléments inconnus. Il faut toujours introduire les variables que l'on utilise par la suite.

1.4 ... traduite dans un langage restreint ...

Si l'on parvient à faire en sorte qu'une machine comprenne les phrases écrites par un utilisateur en langue naturelle, on peut lui demander de traduire ce qu'elle a compris dans un autre langage, plus formel, interprétable par un programme dont le but est de valider les preuves. Au lieu de valider immédiatement la preuve écrite par l'utilisateur, il vaut mieux partager les difficultés, et passer par une étape intermédiaire entre l'analyse de la preuve et sa validation. Certes on peut penser qu'il y a de la perte d'information à chaque fois que l'on passe par un intermédiaire. Mais cela permet de développer des modules assez fortement indépendants les uns des autres et ainsi de pouvoir améliorer chacune des étapes sans avoir à modifier les autres.

1.4.1 Les outils

Il y a un certain nombre d'outils, deux pour être précis, qui se sont présentés aux participants du projet comme pouvant être des moyens d'analyser les textes mathématiques écrits dans une langue naturelle et de traduire ceux-ci dans un autre langage. Il n'y a aucun outil spécifique à ce domaine mais certains outils existants pourraient être adaptés à ce domaine précis des mathématiques.
Le premier d'entre eux est les ACGs (Abstract Categorial Grammars ou grammaires catégorielles abstraites). Le second outil est la SDRT (Segmented Dicourse Representation Theory ou Théorie des Représentations de Discours Segmentées en français).
Nous souhaitons apporter ici de simples descriptions et des références que le lecteur pourra lire pour de plus amples précisions. Nous donnerons cependant plus de détail sur les ACGs, puisqu'une partie du travail effectué dans cette thèse (cf. chapitre 3) est en relation avec elles.

ACG

Les ACGs (cf. [DeG]) sont un formalisme qui a pour but la description de la syntaxe et de la sémantique de la langue naturelle. Ce formalisme est basé sur le fragment implicatif de la logique linéaire. Ainsi les ACGs engendrent des langages de λ-termes linéaires.
L'idée principale des ACGs est la présence de deux structures, l'une appelée signature abstraite et l'autre signature objet, et d'une application appelée lexique, qui traduit la signature abstraite vers la signature objet. Les deux signatures sont dans le même formalisme : ses éléments sont des λ-termes linéaires typés. Ce qui différencie les deux signatures sont les constantes qu'elles contiennent, ainsi que les types atomiques.
Un lexique est la donnée de l'image des constantes et des types atomiques de la signature abstraite dans la signature objet. Le lexique peut alors être vu comme un morphisme car il est prolongé à tout terme et tout type et doit vérifier une contrainte de bon comportement, à savoir que l'image d'un terme t doit être typable de type l'image du type de t.

Une troisième chose est nécessaire pour définir une ACG, il s'agit d'un type atomique distingué S de la signature abstraite. Cela permet alors de définir deux langages, le langage abstrait étant l'ensemble des termes de la signature abstraite ayant pour type S, et le langage objet étant l'ensemble des termes t tels qu'il existe un terme du langage abstrait dont l'image est t. Si l'on peut traduire le langage abstrait vers le langage objet, on peut vouloir chercher l'ensemble des termes abstraits qui ont pour image un terme donné du langage objet. Le fait que les deux signatures soient dans le même formalisme entraine une facilité d'utilisation. Il est alors possible de composer les ACGs, ce qui signifie que l'on peut considérer toute signature comme objet ou abstraite.
Par exemple la syntaxe concrète peut être une signature objet, la syntaxe abstraite étant la signature abstraite. Puis l'on peut considérer la structure sémantique comme signature objet de la syntaxe abstraite. On obtient donc la sémantique d'une phrase en deux étapes.

SDRT

La SDRT (c.f. [AsBuVi]) est une approche pour l'interprétation du discours qui présente des avantages pour le traitement automatique des langues. Elle a été conçue au départ par Asher en 1993 comme une extension de la DRT (Discourse Representation Theory ou Théorie de la représentation du discours) de Hans Kamp. La DRT a pour but d'interpréter semantiquement les discours (et non seulement les phrases comme c'est le cas dans la sémantique de Montague) avec des expressions dans un langage logique qui représente leur sens. En général ce langage logique est en fait celui de la logique du premier ordre.
La recherche en linguistique et en linguistique computationnelle a mis en évidence que le discours n'est pas une simple séquence de phrases, mais une structure très élaborée, d'où l'extension de la DRT vers la SDRT.
La SDRT envisage d'analyser la structure formelle du discours des points de vue combinés de la tradition de la sémantique formelle en philosophie du langage et de la tradition de l'analyse du discours en linguistique et en linguistique computationnelle.
Dans la SDRT chaque proposition a une ou plusieurs fonctions rhétoriques au sein du discours. Celles-ci ont des effets sémantiques notamment dans les domaines suivants :
– résolution des anaphores ;
– présupposition ;
– résolution d'ambiguïtés lexicales ;
– l'analyse de la quantification plurielle ;
– le calcul des implicatures et des buts des agents dans un dialogue.
En plus de ces aspects théoriques, la SDRT a été conçue pour l'implémentation d'applications en compréhension ou génération de textes. Elle a pour but entre autres de construire la forme logique d'un discours, ce qui rend donc cet outil applicable dans le cas des preuves mathématiques.

1.4.2 Le schéma général de l'analyse à la traduction

L'idée générale du fonctionnement du système depuis l'analyse jusqu'à la traduction est la suivante :
Les ACGs prennent en entrée une phrase de la preuve écrite en langue naturelle. Elles analysent la phrase, et doivent en donner une analyse syntaxique, puis à partir de cette dernière renvoie une analyse sémantique.
L'analyse sémantique est donnée sous forme de DRS (le système de la DRT) en entrée au système SDRT, qui fournit une représentation de la démonstration dans sa globalité, la SDRS. Dans cette représentation, les anaphores sont résolues. Cette phase est appelée analyse pragmatique.
La représentation en SDRS obtenue, celle-ci est transformée pour donner des commandes de l'assistant de démonstration choisi, dans notre cas le langage restreint.

1.4.3 Les propriétés du langage restreint

Le langage restreint doit être un bon intermédiaire. Il doit suivre une grammaire formelle précise pour être interprétable simplement et avoir un vocabulaire suffisant pour pouvoir décrire le mieux possible la preuve écrite par l'utilisateur.
Le langage doit aussi pouvoir admettre des indices sur la manière de faire la preuve. En effet, une preuve faite en langue naturelle, si elle est détaillée et se veut être compréhensible, explique pourquoi telle ou telle étape de la démonstration est correcte, en évoquant éventuellement des hypothèses ou des lemmes précédents.
Ces indications étant précieuses pour un humain lorsqu'il lit une preuve, elles le sont d'autant plus pour une machine qui, si elle est capable de faire beaucoup plus de calculs en un temps donné qu'un humain, doit être guidée correctement pour trouver rapidement la solution sous peine de se perdre dans des calculs inutiles.
Le langage doit pouvoir coller le plus possible à une preuve naturelle dans son cheminement. Mais cela ne doit pas pour autant impliquer une traduction phrase par phrase d'une preuve, qui peut s'avérer moins efficace qu'une traduction d'ensemble, cela par le fait que des informations permettant de valider ce que l'on est en train de faire peuvent ne venir que plus tard. On voit ce genre de phénomène lorsque par exemple on fait une preuve par cas, où l'ensemble des cas n'est donné que petit à petit.
Le fait que ce langage soit proche d'un langage formel signifie qu'il faut pouvoir donner un moyen de valider la preuve qui est faite.
Il faut donc pouvoir distinguer les étapes de preuves, et en tirer des éléments permettant de les justifier par machine. Rien de mieux donc de considérer que le langage décrive des règles logiques, basées par exemple sur de la déduction naturelle, car alors on sait comment valider de telles règles : il suffit de prouver une formule.
Le choix de la déduction naturelle se fait parce que le raisonnement que l'on a lorsque l'on écrit une preuve est naturel, et le formalisme de la déduction naturelle, comme son nom l'indique, a pour but de calquer ce raisonnement dans un système formel.
Une des difficultés est ensuite de donner pour chaque règle décrite une formule adaptée et la plus simple possible. Il y a en effet de manière naïve une formule type qui peut justifier toute règle décrite dans un formalisme tel que la déduction naturelle, mais cette formule peut s'avérer beaucoup trop complexe, car faisant apparaître des éléments inutiles ou recopiés en nombre trop important.
On doit donc attacher à la grammaire du langage restreint une manière d'interpréter les règles décrites en formules les plus fines possibles, ceci afin que la validation, étape nécessaire suivante, soit la plus efficace possible.

1.5 ... puis validée par machine

1.5.1 Un démonstrateur dédié

Le langage restreint étant interprété sous forme de règles logiques, il est nécessaire de valider ces règles. S'il est possible de former une formule qui justifie toute règle décrite dans le langage, il reste alors à prouver que cette formule est vraie. Si le langage est suffisamment libre, c'est à dire que si la description des règles est suffisamment libre on peut imaginer toutes sortes de règles qui sortent du cadre des règles prédéfinies dans les assistants de démonstration usuels. Il est donc nécessaire d'avoir un programme capable de valider ces règles en prouvant les formules qui les justifient. Nous pourrions *a priori* prendre n'importe quel démonstrateur automatique pour cela. Par exemple nous pourrions utiliser la méthode de preuve automatique d'un assistant de démonstration tel que PhoX. C'est d'ailleurs ce qui a été testé dès le début avec le langage new_command.
En effet, le programme PhoX, s'il est éventuellement moins célèbre que les autres cités au tout début de ce chapitre, est développé à Chambéry au laboratoire de mathématiques par Christophe Raffalli. Il est donc l'outil d'aide à la démonstration naturellement utilisé dans le

cadre de cette thèse. Un des buts principaux de cet assistant est qu'il soit autant que possible facile à utiliser et surtout qu'il demande peu de temps d'apprentissage.

Cependant PhoX s'est avéré inefficace, dans certains cas qui paraissaient vraiment évidents, à justifier les étapes de démonstration décrites par les new_commands. Les cas se sont trouvés principalement pour l'application d'une hypothèse par une autre, ou bien pour les règles d'élimination sur une hypothèse. Ce sont donc des cas où la formule courante à prouver n'était pas nécessaire à la validation de la règle.

Dans d'autres cas il pouvait s'agir de règles d'introduction, décrites dans un ordre différent de la formule à prouver, celle-ci pouvant contenir à la fois des $\rightarrow$ et des $\wedge$ à gauche (par exemple $(A \wedge B) \rightarrow C$). Des problèmes de commutativité rendaient alors la preuve très longue.

Une des principales raisons à ces difficultés est certainement que la tactique automatique de PhoX n'est pas faite pour l'utilisation qui en est faite dans notre cas.

Ce que nous voulons également, c'est que le démonstrateur soit capable d'accepter rapidement la preuve donnée. Comme ce qui est demandé par le démonstrateur est la justification de chacune des petites étapes de la preuve, nous pouvons nous dire qu'il est inutile *a priori* d'avoir un démonstrateur d'une efficacité redoutable, car de petites étapes de preuves signifient de petites justifications.

Ainsi il serait bien de créer un démonstrateur qui ferait le plus possible de petits raisonnements, et qui éviterait les raisonnements trop complexes. Un raisonnement trop complexe peut signifier entrer en profondeur dans les formules, ou bien utiliser beaucoup de connaissances, c'est à dire faire un long raisonnement.

La tactique automatique de PhoX a également un autre défaut (si cela en est réellement un), qui est de revenir en arrière en cas d'échec dans la recherche d'une preuve. Ce retour en arrière peut être considéré comme une perte de temps, notamment si certaines choses utiles à la preuve sont oubliées lors du retour en arrière. D'où l'idée d'implémenter un démonstrateur par résolution, dont un des principes est justement de ne faire qu'ajouter des informations, et de ne jamais revenir en arrière.

Enfin le démonstrateur doit être capable de suivre le plus possible les indications données, comme par exemple l'utilisation d'une hypothèse par une autre, pour valider les étapes de démonstration. Ceci est une grande différence avec les démonstrateurs automatiques, dont le but est de prouver une formule quelconque sans avoir pour cela une seule indication.

Il peut certes exister quelques heuristiques portant sur la forme des formules, mais en aucun cas ne sont indiqués de moyens particuliers à chaque formule. L'avantage ici de la description riche d'une preuve est justement de permettre de donner des indications, ce qui peut aider avantageusement la validation.

1.5.2 Dialogue entre l'analyse et la validation

Il est nécessaire d'imaginer qu'il y a une interaction entre l'analyseur et le système qui valide les preuves.

Tout d'abord, l'analyse ayant besoin d'un contexte pour que chaque phrase puisse être comprise, on trouve une partie de ce contexte dans les phrases écrites précédemment, mais également dans le but courant, qui est géré par le système de validation. Ainsi, toutes les hypothèses, les constantes, les variables du but courant, etc... font partie du contexte, ou environnement, dans lequel doit se situer l'analyseur.

D'autre part il est tout à fait possible que l'analyse des preuves puisse donner, à cause d'ambiguïtés non résolues, plusieurs interprétations possibles, et donc plusieurs traductions différentes. Chacune d'entre elles peut alors être testée. Le système de validation doit alors informer l'analyseur du fait que les différentes commandes données sont valides ou non dans le but courant. Si une commande est valide, l'analyseur doit également recevoir le nouveau but courant afin de mettre à jour l'environnement.

1.6 DemoNat comparé aux modes Mizar

Maintenant que nous avons expliqué globalement le fonctionnement du système de DemoNat, nous pouvons pointer les points communs et les différences avec les systèmes utilisant un mode Mizar.
Les points communs sont assez évidents entre le langage restreint du système et le langage Mizar : la syntaxe est assez semblable. Tous deux ont pour but d'être assez proche de la langue naturelle, et d'être compréhensible par tout mathématicien. Un certain nombre de mots sont communs, tels que `let`, `assume`, `proof` et `by`. Il s'agit à chaque fois d'un langage de preuve déclaratif.
Les différences apparaissent dans la sémantique du langage, et donc dans la validation des preuves écrites dans ces langages. Le système Isar, qui est après Mizar l'un des systèmes avec langage de preuve déclaratif les plus développés, fait la différence entre les petites étapes de preuve et les grandes étapes de preuves.
Dans les petites étapes de preuve, il n'est fait aucun appel à un système automatique. C'est seulement pour les grandes étapes, plus complexes, que plusieurs méthodes peuvent être appliquées. Certaines des méthodes de validation de Isar sont assez puissantes, mais il y est fait le moins possible appel.
Notons qu'il en est de même pour Mizar, sauf qu'une seule méthode est appliquée et que le raisonnement de celle-ci est volontairement peu complexe. Il s'agit en fait de raisonnement dans un fragment rapidement décidable de la logique du premier ordre. La raison est que la réponse donnée (commande validée ou non) doit être la plus rapide possible.
C'est en particulier la sémantique du `by` qui est totalement différente entre le système de DemoNat et les modes Mizar. En effet dans notre système `by` permet de modifier l'importance des diverses hypothèses courantes, et donc d'influer sur la preuve automatique, alors que dans les modes Mizar il s'agit de donner précisemment les règles à utiliser, ou bien de donner la méthode de validation à utiliser.
Cela implique que l'utilisateur doit bien connaitre les diverses possibilités offertes dans le cas des systèmes Mizar, alors que dans le système de DemoNat, l'utilisateur a seulement à donner des indications sur la validation de ses étapes de démonstration.
Il s'agit donc *a priori* d'un système plus souple pour l'utilisateur, qui n'a pas à connaître le fonctionnement du système.

Plusieurs critiques peuvent être faites sur le système que nous avons décrit.

- Il n'y a pas de techniques fines pour de petites étapes de démonstration. En effet, presque chaque règle doit être justifiée par le démonstrateur, que la règle soit complexe ou non. L'argument qui peut être donné pour l'utilisation systématique d'un démonstrateur est que les preuves en langue naturelle ne suivent pas exactement la forme des formules prouvées. Ainsi lors de l'introduction des hypothèses, il se peut que l'on permute certaines hypothèses ou certaines variables. Il est alors nécessaire de faire appel à un système automatique permettant de justifier que le nouvel ordre donné respecte bien le but donné.
  Il faut bien voir que le but du projet DemoNat est de rendre plus libre l'écriture de sa preuve, aussi bien dans le langage que dans la structure. Si les langages des modes Mizar sont compréhensibles, ils obligent néanmoins à une rigeur dans l'ordre des arguments donnés et dans la structure de la preuve.
- Le démonstrateur automatique utilisant la résolution, il est impossible d'indiquer des messages d'erreur compréhensibles par des personnes ne connaissant pas cette méthode lorsque la preuve échoue. Le système Mizar a en effet développé un système capable de donner des messages d'erreur précis, permettant de guider l'utilisateur.
  Cet argument est effectivement tout à fait correct en ce qui concerne les démonstrations par résolution, qui transforment complètement les formules données en un ensemble de formules atomiques dont le sens se perd pour toute personne souhaitant comprendre les preuves.
  Il n'est cependant pas interdit de penser que comme le démonstrateur utilisé ici décompose les formules en cours de preuve et non au préalable, il est possible de comprendre les

formules traitées par la machine, et d'en connaître l'origine. C'est à dire que dans notre cas nous avons accès à la décomposition des formules, et dans le cas de preuves assez courtes, on peut retrouver un raisonnement naturel.
Il n'a pas encore été pensé un système d'explication d'échec des preuves de formules données au démonstrateur automatique. On peut malgré tout imaginer un affichage de formules que le démonstrateur n'a pas réussi à démontrer. Il ne faut pas oublier également que la validation de la preuve est la dernière étape d'une chaîne d'analyses qui commence par celle de la preuve en langue naturelle. Ainsi l'analyse du texte peut ne pas avoir résolu certaines anaphores ou ambiguïtés, et il peut être demandé à l'utilisateur plus de précisions.

- Pendant une preuve interactive, l'invocation d'outils de preuve automatique échoue encore et encore, jusqu'à ce que l'utilisateur trouve un manquement dans les hypothèses ou les faits démontrés précédemment, ce qui fait que le succès des preuves automatiques est plus une exception qu'une règle.
  C'est la raison qui fait que le démonstrateur doit véritablement être efficace. D'autre part, l'oubli de certaines hypothèses est effectivement un problème. Par exemple, nous pouvons imaginer une hypothèse de la forme $\exists x \in A, P(x)$. L'utilisateur peut alors écrire, en langue naturelle : "Soit x tel que $P(x)$".
  L'utilisateur a oublié de préciser que x peut être un élément de A, ce qui peut être absolument indispensable à la terminaison de la preuve. On peut alors se demander comment réparer un tel oubli. Une question se pose cependant : doit-on réparer un tel oubli ? Si les outils que nous développons sont pour enseigner la logique aux étudiants, il est certain que de tels choses ne doivent pas être oubliés.
  Si le but est de faire un outil pour des mathématiciens, alors effectivement ce genre de chose devrait pouvoir être détecté. On peut alors se ramener à ce que nous avons dit plus haut, à savoir qu'au moment crucial où la démonstration a besoin de prouver que x appartient à A, la preuve échoue et explique qu'elle n'a pas pu prouver ce fait.

Chapitre 2

Un langage restreint

2.1 Ce que l'on souhaite

Comme nous l'avons écrit dans le chapitre introductif, nous souhaitons un intermédiaire entre la preuve écrite par l'utilisateur et la validation par machine. Cet intermédiaire est un langage restreint dont le but est d'être aisément interprétable et qui permet de décrire la preuve de manière aussi précise que possible.
Pour que ce langage restreint soit raisonnable et interprétable facilement, il doit reposer sur un vocabulaire assez pauvre mais suffisant pour décrire tout ce que l'on veut. Plus on ajoutera de vocabulaire, plus il sera proche de la langue naturelle qui permet la plus libre expression et description des preuves, mais plus on s'éloignera d'un langage de preuve formelle. Il faut donc parvenir à un bon compromis.
Le langage restreint doit pouvoir décrire toutes les preuves que l'on souhaite établir. Or l'interprétation d'une preuve ne se voit pas comme une succession linéaire d'applications de règles. Une preuve naturelle est plutôt une description qui s'interprète comme un arbre. Le langage doit donc pouvoir permettre de décrire des arbres de règles.
Nous devons également pouvoir inclure des indications sur la validité de la preuve, en permettant de préciser l'utilisation d'un lemme, d'une formule par une hypothèse ou avec certaines valeurs. Ces indications sont précieuses pour aider le démonstrateur à orienter ses raisonnements vers la bonne preuve. Ces indications proviennent des preuves écrites en langue naturelle, car une preuve détaillée indique toujours les raisons de la validité du passage d'une étape à une autre.
Nous allons donc voir dans ce chapitre un langage restreint qui tente de répondre à ces exigences. Nous allons en donner la grammaire dans une première section, composée d'une quinzaine de mots clés. La section suivante sera dédiée à l'interprétation d'une phrase de ce langage dans le cadre particulier de la logique du premier ordre.

2.2 La grammaire

Donnons ici la grammaire du langage restreint. Dans une première partie nous décrirons la manière d'introduire des noms et des hypothèses, par liste éventuellement. Dans une seconde nous donnerons la grammaire du langage proprement dit. Nous donnerons dans une dernière partie un exemple simple d'utilisation de ce langage.
Dans cette grammaire les mots-clés (symboles terminaux) de la grammaire sont écrits en majuscule. Les symboles non terminaux sont eux écrits en minuscule.

2.2.1 introduction de noms et d'hypothèses

- Certains mots-clés ont des types particuliers, c'est à dire que l'analyseur lexical doit leur attacher un élément d'un certain type qui sera donné à l'analyseur syntaxique. Voici la liste des mots clés avec arguments :

– NAME : nom de variable ou de constante ;
– FORMULA : formule ;
– HYPNAME : nom d'hypothèse ;
– EQUAL : un nom de variable égale un terme ;
– LABEL : nom de but.

Les autres mots clés sont donnés par l'analyseur quand il détecte le mot clé concret. Par exemple une virgule est analysée comme VIRG.

- Voici les premiers éléments de la grammaire, qui définissent des listes :

```
namelist ::=
         |  NAME
         |  NAME VIRG namelist
form     ::=
         |  FORMULA HYPNAME
         |  FORMULA
formlist ::=
         |  form
         |  form AND formlist
name_form_eq ::=
         |  HYPNAME
         |  FORMULA
         |  EQUAL
name_form_eq_list ::=
         |  name_form_eq
         |  name_form_eq VIRG name_form_eq_list
```

- Maitenant nous pouvons commencer à donner ce qui permet d'introduire de nouveaux noms :

```
letsuite    ::= LET namelist
searchsuite ::= SEARCH namelist
assumesuite ::= ASSUME formlist
deducesuite ::= DEDUCE formlist
```

Les éléments introduits par LET sont des constantes, et celles introduites par SEARCH sont des variables. Les formules données par ASSUME sont des formules que nous ajoutons, et celles données par DEDUCE sont des formules qui doivent être prouvées automatiquement pour pouvoir être ajoutées.

- Voici ce qui permet d'ajouter des indices de preuves susceptibles d'aider le démonstrateur :

```
withsuite    ::= WITH name_form_eq_list
name_or_form ::=
             |  HYPNAME
             |  FORMULA
elwithelt    ::=
             |  name_or_form
             |  name_or_form withsuite
elwithlist   ::=
             |  elwithelt
             |  elwithelt AND elwithlist
bysuite      ::=
             |  BY elwithlist
             |  BY elwithlist bysuite
```

la syntaxe est donc une suite de BY [...] WITH [...]. Après un BY on peut donner le nom d'une hypothèse ou une formule, et après un WITH on peut donner la même chose, plus une égalité de la forme variable=terme.

2.2.2 La partie principale

Voici maintenant la partie principale de la grammaire. L'analyseur syntaxique cherche à reconnaitre un élément de type nc. Autrement dit, nc est l'axiome de la grammaire. Celui-ci a des règles de production qui dépendent de ncs. Les *ncs* décrivent des commandes simples, essentiellement sans indication de preuve. Les règles de production des *ncs* dépendent quant à elles de meta, dont le but principal est de permettre de décrire des règles par les prémisses.

- Les règles de production des *nc* sont les suivantes :

nc ::=
| ncs
| bysuite ncs
| PROVE form
| PROVE form nc
| PROVE form nc IN nc
| PROVE form IN nc
| BYABSURD
| BYABSURD HYPNAME
| BYABSURD nc
| BYABSURD HYPNAME nc
| SET EQUAL
| SET EQUAL nc
| LABEL HYPNAME

Nous verrons ncs plus bas, il s'agit de commandes simples, sans les indices de preuve donnés par BY [...] WITH [...]. D'autres commandes n'ont pas besoin d'indice :
- PROVE permet d'introduire un résultat intermédiaire à prouver ;
- BYABSURD permet de faire le raisonnement par l'absurde ;
- SET permet de donner une valeur à une variable de manière impérative. La notation est quelque chose semblable à 'set $x = f(t)$' où x est une variable du but courant ;
- LABEL permet de donner un nom au but courant.

Remarque 2.2.1 Pour chaque mot clé il peut y avoir plusieurs entrées car l'utilisateur peut choisir de continuer les branches de l'arbre de preuve ou non, et peut choisir quelles branches de preuves il continue.
- "PROVE form nc IN nc", qui est le plus général des PROVE, est à interpréter ainsi : le premier *nc* est la commande qui correspond à la preuve de la formule form et le second est la preuve de la formule but courante avec form ajoutée aux hypothèses.
- Pour BYABSURD on peut décider ou non de donner un nom à la négation de la formule but courante avec HYPNAME, puis de faire tout de suite la preuve ou non avec nc.

Remarque 2.2.2 Toutes les variables introduites, sauf si n'importe quelle valeur de ces variables rend le but courant valide, doivent être instanciées par la commande SET pour que la preuve termine. Nous verrons toutefois au chapitre 4, à la section 4.4.4, comment il pourrait être possible de faire découvrir la valeur d'une variable de manière automatique.

Note 2.2.3 le mot clé LABEL peut-être utilisé dans le cadre d'un assistant de démonstration, où des commandes indépendantes du langage restreint permettent de changer le but courant. Nous n'avons en effet pas donné dans la grammaire du langage restreint la possibilité de commencer la preuve d'un but qui n'est pas le but courant. Cela permettait de simplifier la grammaire et son interprétation. D'autre part changer de but courant est un artifice qui ne fait pas avancer la preuve.

- Les règles de production des *ncs* sont les suivantes : :

```
ncs  ::=
       |  deducesuite nc
       |  deducesuite
       |  TRIVIAL
       |  meta
```

On peut ainsi au choix :
- déduire un certain nombre d'hypothèses avec deducesuite puis éventuellement continuer la preuve, d'où la présence de nc ;
- considérer que le but courant doit être démontré automatiquement avec TRIVIAL ;
- décrire une règle plus complexe avec meta.

• Voici les règles de production des *meta* :

```
meta  ::=
        |  letsuite meta
        |  searchsuite meta
        |  assumesuite meta
        |  assumesuite
        |  assumesuite LABEL HYPNAME
        |  SHOW FORMULA nc SHOWN
        |  SHOW FORMULA
        |  PROOF nc ENDPROOF
        |  meta THEN meta
        |  PBEGIN meta PEND
```

meta permet de décrire une règle par les prémisses et donc par les hypothèses supplémentaires que l'on ajoute. Il n'y a aucune commande pour éliminer des hypothèses, donc les prémisses contiennent nécessairement les hypothèses du but courant.
Le choix a été fait qu'aucune hypothèse ne puisse être éliminée car dans une preuve écrite en langue naturelle, il est très rarement (pour ne pas dire jamais) fait mention du fait qu'une hypothèse ne sert plus.
- les LET dans letsuite introduisent des constantes ;
- les SEARCH dans searchsuite introduisent des variables ;
- les ASSUME dans assumesuite permettent d'ajouter des hypothèses à la prémisse que l'on décrit ;
- SHOW permet de changer la formule à prouver, c'est à dire que la prémisse que l'on définit a une formule but différente de la formule but courante ;
- THEN permet de définir la prémisse suivante ;
- PROOF [...] ENDPROOF et PBEGIN [...] PEND sont des parenthèses. Il y a une distinction importante entre les deux. Les premières servent à décrire la preuve de la prémisse que l'on vient de décrire, alors que les secondes permettent de mettre des parenthèses entre prémisses.
  En fait THEN est considéré comme un connecteur binaire qui est associatif à droite, et PBEGIN et PEND sont des parenthèses permettant de faire les associations à gauche.
  Ces dernières parenthèses permettent de mettre en commun des variables,
  constantes ou hypothèses sur plusieurs prémisses et donc d'éviter de les répéter.

Remarque 2.2.4 Il est important de noter que c'est bien un arbre de règles que l'on décrit, et que celui-ci peut être arrêté quand l'utilisateur le désire. Il est alors possible de faire des preuves en une seule commande, qui désigne un arbre de règles. Cela ne pénalise pas du tout la validation des preuves, puisque ce qui est prouvé est la validité de chacune des règles successivement et non toutes les règles globalement comme nous le verrons plus tard.

2.2.3 Un exemple

Nous pouvons donner quelques phrases de preuves en exemple :

Supposons que nous avons à prouver la formule propositionnelle suivante :

$$(a \to c) \to (b \to c) \to (a \vee b) \to c$$

Nous pouvons alors écrire :

1. assume $a \to c$ and $b \to c$ and $a \vee b$ show c.
2. assume a then assume b.
3. deduce c.
4. trivial.
5. deduce c.
6. trivial.

que nous pouvons pour l'instant expliquer ainsi étape par étape :

1. trois hypothèses sont ajoutées, $a \to c$, $b \to c$ et $a \vee b$ et le but est changé en c.
2. on fait une preuve avec deux cas, donc nous obtenons deux buts à prouver. Un premier dans lequel on a a comme hypothèse, et un second dans lequel on a b comme hypothèse.
3. Dans le premier but, où l'on a supposé a, on déduit c.
4. toujours dans le premier but, celui-ci est évident et on termine sa preuve.
5. Dans le second but, où l'on a supposé b, on déduit c.
6. Le second but est évident et on termine la preuve. Il ne reste alors plus rien à prouver.

Nous pouvons aussi faire la preuve en une seule longue commande :

assume $a \to c$ and $b \to c$ and $a \vee b$ show c assume a proof deduce c trivial endproof then assume b proof deduce c trivial endproof shown.

Nous remarquons ici l'utilisation de proof et endproof.

Remarque 2.2.5 Il est bien certain qu'une phrase longue décrivant un arbre complexe est bien moins lisible qu'une succession de phrases courtes ne décrivant qu'une seule règle à chaque fois. Cependant il faut bien garder en mémoire que ce langage restreint est supposé être un intermédiaire entre la preuve en langue naturelle et la validation par la machine. Donc à aucun moment l'utilisateur n'est censé la lire.
Il est cependant tout à fait possible de faire des preuves en utilisant ce langage, et l'on peut faire le choix de faire soit des petites phrases soit de grandes.

Le lecteur intéressé pourra voir en annexe IV d'autres exemples de l'usage du vocabulaire restreint.

2.3 L'interprétation et la validation

Si le langage en lui-même est indépendant de la logique, son interprétation dépend de la logique utilisée. Ce que l'on interprète est en fait une phrase écrite dans le langage restreint dans un contexte, c'est à dire que l'on se donne en plus de la phrase le but courant. La phrase est aussi appelée commande.
Nous considérons dans cette partie une logique du premier ordre, et les buts courants sont des séquents pour la déduction naturelle, donc le but courant a la forme $\mathcal{H} \vdash G$, où $\mathcal{H}$ est un ensemble de formules, et G est une formule.

L'interprétation d'une commande dans un contexte donné se fait en deux étapes.
– La première est la détermination d'un arbre de buts étiqueté par les règles décrites par la commande ainsi que par les indices de preuve, que nous appellerons l'arbre de preuve. Certaines conditions doivent être vérifiées lors de la construction de cet arbre.

– La seconde est la détermination des formules à prouver pour justifier chacune des règles de l'arbre. Pour cette seconde étape, presque chaque règle de l'arbre de preuve est associée à une formule indépendante du reste de l'arbre. Ce n'est pas le cas pour toutes les règles car certaines restent toujours valides et ne nécessitent pas de preuve. C'est le cas principalement de la règle de coupure introduite par la commande PROVE et de BYABSURD.

Nous simplifierons nos explications par rapport à ce qui a été implémenté afin de clarifier au mieux les choses. En effet, les deux étapes que nous venons de décrire ne sont dans les faits pas totalement indépendantes, mais décrire le tout en même temps serait sans doute lourd et prêterait à confusion.

L'arbre de preuve est ensuite parcouru lors de la validation de la commande. La validation utilise les indices de preuves, et peuvent ajouter des étapes supplémentaires aux règles déduites quand les indices doivent être justifiés eux-mêmes. L'appel au démonstrateur se fait alors pour prouver

– les formules qui permettent de valider les règles ;
– des formules introduites par deducesuite ou par les BY[...] WITH[...] ;
– le but courant lui-même lors d'un appel à TRIVIAL.

2.3.1 L'arbre de preuve

Définissons récursivement les fonctions

– $interp_{nc}$ qui à une commande nc et un but courant $B = (H \vdash G)$ associe un arbre de preuve ;
– $interp_{ncs}$ qui à une commande simple ncs et un but courant $B = (H \vdash G)$ associe un arbre de preuve ;
– $interp_{meta}$ qui à une commande $meta$ et un but courant $B = (H \vdash G)$ associe une liste (ordonnée) d'arbres de preuve, dont la racine de chacun est la prémisse d'une règle.

Les arbres seront ici étiquetés par des mots clés : **stop**, **trivial**, **valide**, **prouve** et **deduit** qui signifient comment doit être éventuellement validée la règle. Les mots clés **stop** et **trivial** étiquettent les feuilles et les autres étiquettent les noeuds. Voici leur signification :

– **stop** : le but est laissé à l'utilisateur ;
– **trivial** :le but doit être prouvé automatiquement ;
– **deduit** : chacune des formules ajoutées en hypothèse doit être prouvée sous les hypothèses courantes ;
– **valide** : la règle est valide et n'a besoin d'aucune justification ;
– **prouve** : la règle doit être justifiée par un démonstrateur en prouvant une formule, que nous déterminerons dans la prochaine sous-section.

Pour ce qui est des buts courants, nous considérons en plus qu'à chacun est associé l'ensemble de ses constantes et variables libres, non nécessairement présentes dans le but, mais ordonnées selon leur ordre d'introduction.

Nous définissons les fonctions sur les cas les plus généraux, "PROVE form" étant un cas particulier de "PROVE form nc IN nc" par exemple. Nous avons alors besoin d'ajouter les commandes vides, ce qui permet de voir "PROVE form" comme "PROVE form $\emptyset$ IN $\emptyset$".

1. pour $interp_{nc}$:
   – $interp_{nc}(\emptyset\,;B)=$

$$\frac{}{B}\,\textbf{stop}$$

   – $interp_{nc}(\text{ncs}\,;B)=interp_{ncs}(\text{ncs}\,;B)$
   – $interp_{nc}(\text{bysuite ncs}\,;B)=interp_{ncs}(\text{ncs}\,;B)$
   – $interp_{nc}(\text{PROVE } F \text{ nc}_1 \text{ IN nc}_2\,;B)=$

$$\frac{interp_{nc}(\text{nc}_1; H \vdash F) \quad interp_{nc}(\text{nc}_2; H, F \vdash G)}{G}\,\textbf{valide}$$

– $interp_{nc}$(BYABSURD HYPNAME nc ;B)=

$$\frac{interp_{nc}(\text{nc}; H, \neg G \vdash \bot)}{G}\ \textbf{valide}$$

– $interp_{nc}$(SET EQUAL nc ;B)=

$$\frac{interp_{nc}(\text{nc}; B\sigma)}{B}\ \textbf{valide}$$

Où σ est la substitution qui transforme la variable donnée par le premier membre de EQUAL en le terme donné par le second membre de EQUAL. Cette commande est valide si la variable x dont on veut donner une valeur est une variable libre du but et que le terme n'utilise que les variables et constantes de B introduites avant x. Ensuite la variable n'est plus libre dans le but $B\sigma$.

– $interp_{nc}$(LABEL HYPNAME ;B)=

$$\frac{}{B}\ \textbf{stop}$$

2. pour $interp_{ncs}$:

– $interp_{ncs}$(deducesuite nc ;B)=

$$\frac{interp_{nc}(\text{nc}; H, F_1, \ldots, F_n \vdash G)}{B}\ \textbf{deduit}$$

Si la liste de formules donnée après DEDUCE est $F_1, \ldots, F_n$.

– $interp_{ncs}$(TRIVIAL,B)=

$$\frac{}{B}\ \textbf{trivial}$$

– $interp_{ncs}$(meta ;B)=

$$\frac{interp_{meta}(\text{meta}; B)}{B}\ \textbf{prouve}$$

$interp_{meta}$ étant une liste d'arbres, on voit la règle comme ayant pour prémisses l'ensemble des racines des arbres.

3. pour $interp_{meta}$:

– $interp_{meta}$(letsuite meta ;B)=$interp_{meta}$(meta ;B)

Où les constantes introduites dans le "letsuite", qui ne doivent pas être des constantes ou des variables au moment de leur introduction, deviennent des constantes de B.

– $interp_{meta}$(searchsuite meta ;B)=$interp_{meta}$(meta ;B)

Où les variables introduites dans le "searchsuite", qui ne sont pas des constantes ou variables au moment de leur introduction, deviennent des variables de B.

– $interp_{meta}$(assumesuite meta ;B)= $interp_{meta}$(meta ;$H, F_1, \ldots, F_n \vdash G$)

Où F_1,...,F_n sont les formules introduites dans le "assumesuite".

– $interp_{meta}$(SHOW F nc SHOWN ;B)=$interp_{nc}$(nc ;$H \vdash F$)

– $interp_{meta}$(PROOF nc ENDPROOF ;B)=$interp_{nc}$(nc ;B)

– $interp_{meta}$(meta$_1$ THEN meta$_2$;B)=

$$interp_{meta}(\text{meta}_1 ; B)\ @\ interp_{meta}(\text{meta}_2 ; B)$$

Où @ représente la concaténation des deux listes.

– $interp_{meta}$(PBEGIN meta PEND ;B)=$interp_{meta}$(meta ;B)

Remarque 2.3.1 Les parenthèses PBEGIN et PEND ne servent en fait qu'à différencier les différents cas amenés par THEN, ce qui fait que l'interprétation de ces parenthèses est aussi simple. Il est toutefois important de noter qu'il est possible de donner des hypothèses, variables ou constantes en commun à plusieurs prémisses successives.

Il y a un certain nombre de contraintes :
- Les variables et constantes introduites doivent être fraîches dans le but courant, ce que nous avons déjà expliqué ;
- Les hypothèses qui sont nommées par les BY et WITH doivent appartenir au but courant.

Exemple 2.3.2 Reprenons l'exemple que nous avons vu précedemment. Nous rappelons que nous avons à prouver la formule propositionnelle

$$F = (a \to c) \to (b \to c) \to (a \vee b) \to c$$

Dans le cas où l'on fait plusieurs commandes, voyons les interprétations.
- "assume $a \to c$ and $b \to c$ and $a \vee b$ show c" : il s'agit d'une commande meta, ce qui justifie les deux premières égalités

$$interp_{nc}(\text{assume } a \to c \text{ and } b \to c \text{ and } a \vee b \text{ show } c\,;\ \emptyset \vdash F)$$
$$= interp_{ncs}(\text{assume } a \to c \text{ and } b \to c \text{ and } a \vee b \text{ show } c\,;\ \emptyset \vdash F)$$
$$= \frac{interp_{meta}(\text{assume } a \to c \text{ and } b \to c \text{ and } a \vee b \text{ show } c;\ \emptyset \vdash F)}{\vdash F}\ \textbf{prouve}$$
$$= \frac{interp_{meta}(\text{show } c; a \to c, b \to c, a \vee b \vdash F)}{\vdash F}\ \textbf{prouve}$$
$$= \frac{\dfrac{}{a \to c, b \to c, a \vee b \vdash c}\ \textbf{stop}}{\vdash F}\ \textbf{prouve}$$

- "assume a then assume b" est également une commande meta :

$$interp_{nc}(\text{assume } a \text{ then assume } b\,; a \to c, b \to c, a \vee b \vdash c)$$
$$= \frac{interp_{meta}(\text{assume } a \text{ then assume } b; a \to c, b \to c, a \vee b \vdash c)}{a \to c, b \to c, a \vee b \vdash c}\ \textbf{prouve}$$
$$= \frac{interp_{meta}(\text{assume } a; ..., a \vee b \vdash c) \quad interp_{meta}(\text{assume } b; ..., a \vee b \vdash c)}{a \to c, b \to c, a \vee b \vdash c}\ \textbf{prouve}$$
$$= \frac{\dfrac{}{a \to c, b \to c, a \vee b, a \vdash c}\ \textbf{stop} \qquad \dfrac{}{a \to c, b \to c, a \vee b, b \vdash c}\ \textbf{stop}}{a \to c, b \to c, a \vee b \vdash c}\ \textbf{prouve}$$

- deduce c : est une commande ncs

$$interp_{nc}(\text{deduce } c\,; a \to c, b \to c, a \vee b, a \vdash c)$$
$$= \frac{interp_{nc}(\emptyset; a \to c, b \to c, a \vee b, a, c \vdash c)}{a \to c, b \to c, a \vee b, a \vdash c}\ \textbf{deduit}$$
$$= \frac{\dfrac{}{a \to c, b \to c, a \vee b, a, c \vdash c}\ \textbf{stop}}{a \to c, b \to c, a \vee b, a \vdash c}\ \textbf{deduit}$$

- trivial :

$$interp_{nc}(\text{trivial}\,; a \to c, b \to c, a \vee b, a, c \vdash c)$$
$$= \frac{}{a \to c, b \to c, a \vee b, a, c \vdash c}\ \textbf{trivial}$$

- deduce c : semblable au deduce précédent.
- trivial : semblable au trivial précédent.

Exemple 2.3.3 Dans le cas où la preuve est faite en une seule longue commande, on peut vérifier que l'interprétation de
assume $a \to c$ and $b \to c$ and $a \vee b$ show c assume a proof deduce c trivial endproof then assume b proof deduce c trivial endproof shown
est bien :

$$\dfrac{\dfrac{\dfrac{}{a \to c, b \to c, a \vee b, a, c \vdash c}\ \textbf{trivial}}{a \to c, b \to c, a \vee b, a \vdash c}\ \textbf{deduit} \qquad \dfrac{\dfrac{}{a \to c, b \to c, a \vee b, b, c \vdash c}\ \textbf{trivial}}{a \to c, b \to c, a \vee b, b \vdash c}\ \textbf{deduit}}{a \to c, b \to c, a \vee b \vdash c}\ \textbf{prouve}$$
$$\dfrac{a \to c, b \to c, a \vee b \vdash c}{\vdash F}\ \textbf{prouve}$$

2.3.2 Formule de validation d'une règle

C'est seulement pour les règles de l'arbre de preuve étant étiquetées par **prouve** que nous devons donner une formule qui valide les règles. En effet, les autres les règles sont toujours valides, ou bien pour **deduit** nous avons expliqué que chacune des formules ajoutées doit être prouvée à partir des hypothèses courantes.
Très globalement, si la règle a la forme

$$\dfrac{H, H_1 \vdash G_1 \quad \dots \quad H, H_n \vdash G_n}{H \vdash G}$$

Alors il y a une formule qui justifie cette règle :

$$(H \to H_1 \to G_1) \to \cdots \to (H \to H_n \to G_n) \to H \to G$$

Or cette formule est loin d'être très efficace pour ce que nous faisons. Comme les hypothèses H du but courant ne sont jamais effacées, nous pouvons déjà changer la formule en :

$$(H_1 \to G_1) \to \cdots \to (H_n \to G_n) \to H \to G$$

Ensuite il arrive que la formule à prouver n'ait pas changé dans la règle qui est décrite par la commande, c'est à dire que $G_1 = \cdots = G_n = G$). Dans ce cas il paraît inutile d'utiliser autant de fois la formule G dans la formule qui la justifie.
D'autre part, on souhaite pouvoir faciliter la tâche du démonstrateur le plus possible. Ainsi on peut essayer de décomposer la formule en un ensemble d'hypothèses et une formule à prouver plus petite, ce qui fait que l'on ne prouve pas une formule mais un but.
Il faut aussi essayer autant que possible de regrouper les formules contenant une même variable introduite par la commande, cela afin de ne faire qu'une liaison de la variable par un quantificateur, ce qui réduit les décompositions possibles, et respecte également la description donnée par la commande.
Avant de continuer, nous devons définir l'interprétation restreinte aux meta-règles du langage. En effet nous souhaitons donner une formule pour les règles à justifier, et cette formule ne dépend pas du reste de l'arbre. Ainsi pour simplifier les choses nous pouvons considérer une version simplifiée des règles de production de meta. Nous aurons ensuite besoin de cette interprétation lorsque nous devrons donner une formule de validation.

Définition 2.3.4 *Les meta-règles sont définies par la grammaire suivante :*

$$\begin{aligned} meta ::= {} & let(x, meta) \mid search(x, meta) \mid assume(H, meta) \mid show(K) \\ & \mid then(meta, meta) \mid proof \end{aligned}$$

Définition 2.3.5 *L'interprétation d'un but courant $H \vdash G$ et d'une méta-règle est définie par induction. En considérant que la formule but courante sera toujours notée G, nous noterons simplement $\mathcal{I}(H, meta)$ au lieu de $\mathcal{I}(H \vdash G, meta)$*

– $\mathcal{I}(H, let(x, meta)) = \mathcal{I}(H, meta)$;
– $\mathcal{I}(H, search(x, meta)) = \mathcal{I}(H, meta)$;
– $\mathcal{I}(H, assume(F, meta)) = \mathcal{I}(F :: H, meta)$;
– $\mathcal{I}(H, show(K)) = H \vdash K$;
– $\mathcal{I}(H, proof) = H \vdash G$;
– $\mathcal{I}(H, then(meta_1, meta_2)) = \mathcal{I}(H, meta_1) \cup \mathcal{I}(H, meta_2)$.

Définition 2.3.6 *Si une méta-règle ne contient pas de show, on dit que le but courant est resté inchangé. Sinon on dit que le but courant est changé.*

Remarque 2.3.7 Le lecteur pourra se convaincre qu'il s'agit de l'interprétation simplifiée déjà donnée plus haut des commandes *meta*. La simplification porte principalement sur le fait que l'on ne peut pas continuer la description de l'arbre de preuve avec proof et show. D'autre part le problème d'associativité des parenthèses PBEGIN [...] PEND disparait grâce à l'écriture des meta comme des termes, où en particulier then est une fonction à deux arguments.

Définition 2.3.8 *Nous appelons méta-variable un symbole introduit par search. Nous appelons variable un symbole introduit par let.*

Note 2.3.9 la définition 2.3.8 donne une vision différente de celle donnée dans la section précédente. Jusqu'à présent nous parlions de constante pour les symboles introduits par *let* et de variables pour les symboles introduits par *search*. En effet l'utilisateur a comme information qu'il traite de constantes (qui n'ont pas le droit d'être instanciées) et de variables (auxquelles on peut donner des valeurs). Pour ce qui est du vocabulaire de la machine, nous préférons parler de variables, et de méta-variables lorsque celles-ci représentent un terme.

Les méta-variables sont destinées à être remplacées par des termes. Ainsi il est possible de donner une valeur à une méta-variable, c'est à dire d'appliquer une substitution σ, dont le domaine est un ensemble de méta-variables. Cette substitution doit toutefois vérifier certaines contraintes que nous donnons :

Définition 2.3.10 *Une substitution vérifie les contraintes si elle vérifie les points suivants :*
– *si une méta-variable x est dans le domaine de σ, alors x n'apparaît pas dans l'image de σ ;*
– *L'image d'une variable x par σ est un terme n'utilisant que les variables (méta-variables ou non) introduites avant.*

Note 2.3.11 La première contrainte dans la définition 2.3.10 signifie qu'une fois que la valeur d'une variable est donnée, aucune occurrence de cette variable ne doit subsister.
La seconde contrainte implique qu'en aucun cas une méta-variable ne peut être remplacée par un terme contenant une variable introduite après. Cette contrainte est naturelle car la méta-variable représente un terme, et au moment où elle est introduite les variables futures sont inconnues.
La définition reste informelle du fait que nous n'avons pas défini ce que signifie qu'une variable soit introduite avant ou après une autre. Ceci se lit dans l'interprétation des méta-règles si l'on ajoute à chaque *search* et *let* les variables au but courant, en conservant un ordre de leur apparition. Si l'on considère l'interprétation sous forme d'arbre (l'interprétation du *then* permettant de multiples branchement, les autres étant linéaires) l'ordre d'introduction de constantes et de variables se lit dans leur succession sur la branche partant de la racine et allant au but choisi.

Remarque 2.3.12 Dans chaque prémisse d'une règle les variables peuvent être distinctes, mais il peut également y avoir des variables en commun. Il est à remarquer que si une variable est commune à deux prémisses, alors les variables introduites avant sont également communes.

Nous devons maintenant distinguer le cas où le but courant est changé et le cas où le but courant est inchangé.

Le but courant est changé

S'il existe une prémisse pour laquelle le but est différent, alors la formule est semblable à celle donnée plus haut, avec quelques optimisations supplémentaires :

- Il est possible de faire des introductions sur la formule implicative, et donc de donner le but suivant :

$$H_1 \to G_1, \ldots, H_n \to G_n, H \vdash G$$

  Où l'on considère que H_i est la conjonction des formules ajoutées dans la i-ème prémisse.
- Dans le cas où l'on a introduit des variables et des constantes, il faut les quantifier devant les formules. On peut alors regrouper les formules dans le cas où des variables seraient communes à plusieurs prémisses.

Exemple 2.3.13 imaginons la phrase "let x PBEGIN assume A(x) show C then assume B(x) show D PEND". Alors la règle associée est

$$\frac{H, A(x) \vdash C \quad H, B(x) \vdash D}{H \vdash G}$$

et nous pouvons donner comme but à prouver pour la valider :

$$H, \forall x.((A(x) \to C) \wedge (B(x) \to D)) \vdash G$$

Pour voir cela, il suffit de faire la preuve que $H \vdash G$ est bien dérivable à partir des prémisses du haut de la règle et du but à prouver. Pour cela on peut par exemple utiliser entre autres la règle de coupure dérivable en déduction naturelle :

$$\frac{\Gamma, F \vdash G \quad \Gamma \vdash F}{\Gamma \vdash G}\, cut$$

Donnons une définition un peu plus formelle du but à prouver, et démontrons qu'elle implique bien la validité de la règle.

Définition 2.3.14 *Définissons ϕ_c la formule associée au but courant $H \vdash G$ et à une méta-règle meta par induction. La formule but courante étant toujours notée G et comme de plus les hypothèses H n'apparaissent pas dans la formule nous notons $\phi_c(meta)$ au lieu de $\phi_c(H \vdash G, meta)$:*
- $\phi_c(let(x, meta)) = \forall x.\phi_c(meta)$;
- $\phi_c(search(x, meta)) = \exists x.\phi_c(meta)$;
- $\phi_c(assume(F, meta)) = F \to \phi_c(meta)$;
- $\phi_c(show(K)) = K$;
- $\phi_c(proof) = G$;
- $\phi_c(then(meta_1, meta_2)) = \phi_c(meta_1) \wedge \phi_c(meta_2)$.

Définition 2.3.15 *Etant donnés un but courant $H \vdash G$ et une méta-règle meta, alors le but à prouver pour justifier la règle est $H, \phi_c(meta) \vdash G$.*

En effet, si l'on parvient à prouver que de l'ensemble de séquents $\mathcal{I}(H, meta)$ on peut déduire (en déduction naturelle) $H \vdash \phi_c(meta)$, la coupure permet de conclure :

$$\frac{H, \phi_c(meta) \vdash G \qquad \dfrac{\mathcal{I}(H, meta)}{H \vdash \phi_c(meta)}}{H \vdash G}\, cut$$

Nous prouvons un fait plus général :

Proposition 2.3.16 *Soit $H \vdash G$ le but courant et soit meta une méta-règle. Soit σ une substitution sur les méta-variables qui apparaissent dans $H \vdash G$ ou qui sont introduites par meta vérifiant les contraintes définies à la définition 2.3.10.*
Alors de $\mathcal{I}(H, meta)\sigma$ on peut déduire $(H \vdash \phi_c(meta))\sigma$, où les seules méta-variables libres sont celles de $H \vdash G$ sauf celles appartenant au domaine de σ.

Preuve : Compte tenu de la définition informelle 2.3.10 des contraintes de σ, cette preuve ne peut être qu'informelle. Nous cherchons tout de même à la détailler le plus possible. Elle se fait par induction sur la méta-règle :
- $\mathcal{I}(H, let(x, meta))\sigma = \mathcal{I}(H, meta)\sigma$, d'où l'on déduit par induction $(H \vdash \phi_c(meta))\sigma$. La variable x n'étant pas une méta-variable, elle n'est pas substituée. Elle ne peut pas apparaître dans l'image d'une méta-variable libre, car celles-ci ont été introduites avant x. Comme la variable x n'apparaît pas ailleurs on a donc

$$\frac{(H \vdash \phi_c(meta))\sigma}{(H \vdash \forall x.\phi_c(meta))\sigma}\,\forall_i$$

  Or $\forall x.\phi_c(meta) = \phi_c(let(x, meta))$.
- $\mathcal{I}(H, search(x, meta))\sigma = \mathcal{I}(H, meta)\sigma$, d'où l'on déduit par induction $(H \vdash \phi_c(meta))\sigma$. La variable x est une méta-variable. Si elle appartient au domaine de σ, alors $\sigma(x)$ est un terme n'utilisant pas les variables introduites avant et x n'apparaît pas dans l'image de σ, donc x n'est pas libre dans $(H \vdash \phi_c(meta))\sigma$. On peut alors remplacer le terme $\sigma(x)$ par x et lier par un quantificateur existentiel :

$$\frac{(H \vdash \phi_c(meta))\sigma}{(H \vdash \exists x.\phi_c(meta))\sigma}\,\exists_i$$

  Or $\exists x.\phi_c(meta) = \phi_c(search(x, meta))$.
- $\mathcal{I}(H, assume(F, meta))\sigma = \mathcal{I}(F :: H, meta)\sigma$, d'où l'on déduit par induction $(F, H \vdash \phi_c(meta))\sigma$. Puis par utilisation de la règle $\rightarrow_i$ on obtient $(H \vdash F \rightarrow \phi_c(meta))\sigma$. Or $F \rightarrow \phi_c(meta) = \phi_c(assume(F, meta))$.
- $\mathcal{I}(H, show(K))\sigma = (H \vdash K)\sigma$. Or $\phi_c(show(K)) = K$, donc c'est terminé.
- $\mathcal{I}(H, proof)\sigma = (H \vdash G)\sigma$. Or $\phi_c(proof) = G$, donc c'est terminé.
- $\mathcal{I}(H, then(meta_1, meta_2))\sigma = \mathcal{I}(H, meta_1)\sigma \cup \mathcal{I}(H, meta_2)\sigma$. Des deux ensembles on déduit par induction $(F, H \vdash \phi_c(meta_1))\sigma$ et $(F, H \vdash \phi_c(meta_2))\sigma$. Alors grâce à la règle $\wedge_i$ et à la contraction nous obtenons $(F, H \vdash \phi_c(meta_1) \wedge \phi_c(meta_2))\sigma$. Or $\phi_c(then(meta_1, meta2)) = \phi_c(meta_1) \wedge \phi_c(meta_2)$. □

Remarque 2.3.17 Il est possible d'améliorer encore légèrement le but à prouver. Si la formule $\phi_c(meta)$ est une conjonction $A \wedge B$, alors prouver $H, \phi_c(meta) \vdash G$ est équivalent à prouver $H, A, B \vdash G$. On peut donc faire ainsi pour toute formule conjonctive.

Le but courant reste inchangé

Si aucune prémisse n'a de but différent, alors la formule est différente. La règle est de la forme

$$\frac{H, H_1 \vdash G \quad \ldots \quad H, H_n \vdash G}{H \vdash G}$$

Dans le cas propositionnel, le but à prouver est $H \vdash H_1 \vee \ldots \vee H_n$, c'est à dire qu'avec les hypothèses H on doit pouvoir prouver $H_1 \vee \ldots \vee H_n$. Si H_i est en fait un ensemble de formules, on considère cet ensemble comme la conjonction de ses éléments.
Pour voir cela il suffit d'utiliser les règles suivantes dérivables en déduction naturelle :

$$\frac{\Gamma, A \vdash B \quad \Gamma \vdash A}{\Gamma \vdash B}\,\text{cut} \qquad \frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \vee B \vdash C}\,\vee_g$$

En fait prouver le but $H \vdash H_1 \vee \ldots \vee H_n$ implique que la règle est valide, mais la réciproque est fausse. En effet, le fait que la règle soit valide peut provenir du fait que la formule G soit vraie, et dans ce cas on a $H, \neg G \vdash H_1 \vee \ldots \vee H_n$, la prouvabilité de ce but devenant équivalent à la validité de la règle. Dans notre cas concret, nous avons préféré l'implication à l'équivalence, ce qui permet de gagner sur le raisonnement du démonstrateur en enlevant une formule.
Pour ce qui est du premier ordre, il nous faut traiter du cas des variables et des méta-variables. Les anciennes variables et méta-variables doivent être considérées comme des constantes (nous verrons dans le chapitre 4 à la section 4.4.4 la possibilité de faire autrement pour les variables). Dans le cas où l'on a introduit pour la règle de nouvelles variables et méta-variables il en va autrement. Les variables (introduites par *let*) sont ici quantifiées existentiellement et les méta-variables (introduites par *search*) sont quantifiées universellement.

Exemple 2.3.18 On voit bien que la phrase "let x assume $P(x)$" est validée si l'on peut prouver $\exists x.P(x)$, car la règle associée est la suivante :

$$\frac{H, P(x) \vdash G}{H \vdash G}\ \textbf{prouve}$$

qui est validée par le but $H \vdash \exists x.P(x)$, car c'est exactement la règle de l'élimination du quantificateur existentiel de la déduction naturelle qui nous donne le résultat :

$$\frac{H \vdash \exists x.P(x) \quad H, P(x) \vdash G \quad \text{la variable } x \text{ est fraîche}}{H \vdash G}\ \exists_e$$

Remarquons cependant que si pour la déduction naturelle x est une variable, pour notre interprétation il s'agit d'une constante, car non susceptible d'être modifiée par une substitution. Nous avons déjà vu pour la construction de l'arbre de preuve que nécessairement les noms donnés devaient être frais pour le but courant. La raison en est donc ici donnée.

Exemple 2.3.19 Pour le cas des variables, voici une phrase que l'on peut donner : "search x assume $P(x)$". La règle associée est la suivante :

$$\frac{H, P(?x) \vdash G}{H \vdash G}\ \textbf{prouve}$$

Où $?x$ est ici une méta-variable susceptible d'être instanciée par un terme. Il faut donc voir $?x$ comme un terme. Cette règle est validée par le but $H \vdash \forall x.P(x)$:

$$\frac{\dfrac{H \vdash \forall x.P(x)}{H \vdash P(?x)}\ \forall_e \qquad H, P(?x) \vdash G}{H \vdash G}\ \text{cut}$$

Bien sûr l'ordre de la quantification doit se faire dans l'ordre donné dans la phrase.

Donnons une définition plus formelle du but à prouver, et démontrons qu'elle implique bien la validité de la règle.

Définition 2.3.20 *Définissons ϕ_i la formule associée à but courant $H \vdash G$ et à une méta-règle ne contenant pas de show(K) par induction. En considérant qu'aucune formule de $H \vdash G$ n'apparaît, nous notons simplement $\phi_i(meta)$ au lieu de $\phi_i(G, meta)$:*
- *$\phi_i(let(x, meta)) = \exists x.\phi_i(meta)$ si $\phi_i(meta) \neq \top$, $\top$ sinon ;*
- *$\phi_i(search(x, meta)) = \forall x.\phi_i(meta)$ si $\phi_i(meta) \neq \top$, $\top$ sinon ;*
- *$\phi_i(assume(F, meta)) = H \wedge \phi_i(meta)$ si $\phi_i(meta) \neq \top$, $\top$ sinon ;*
- *$\phi_i(proof) = \top$;*
- *$\phi_i(then(meta_1, meta_2)) = \phi_i(meta_1) \vee \phi_i(meta_2)$, en simplifiant si l'une ou les deux formules sont $\top$.*

Définition 2.3.21 *Etant donnés un but courant $H \vdash G$ et une méta-règle meta qui ne contient pas de show(K), le but à prouver pour justifier la règle est $H \vdash \phi_i(meta)$.*

En effet, si l'on parvient à prouver que de l'ensemble de séquents $\mathcal{I}(H, meta)$ on peut déduire (en déduction naturelle) $H, \phi_i(meta) \vdash G$, la coupure permet de conclure :

$$\frac{H \vdash \phi_i(meta) \quad \dfrac{\mathcal{I}(H, meta)}{H, \phi_i(meta) \vdash G}}{H \vdash G}\, cut$$

Nous prouvons un fait plus général :

Proposition 2.3.22 *Soit $H \vdash G$ le but courant et meta une méta-règle. Soit σ une substitution sur les méta-variables du but courant et introduites par meta vérifiant les contraintes données à la définition 2.3.10.*
Alors de $\mathcal{I}(H, meta)\sigma$ on peut déduire $(H, \phi_i(meta) \vdash G)\sigma$, où les seules méta-variables libres sont celles de $H \vdash G$ sauf celles appartenant au domaine de σ.

Preuve : Compte tenu de la définition informelle des propriétés de σ, cette preuve ne peut être qu'informelle. Nous cherchons tout de même à la détailler le plus possible. Dans cette preuve nous allons utiliser des règles qui sont dérivables en déduction naturelle (le lecteur pourra éventuellement se convaincre en lisant [DaNoRa])

$$\frac{H, F(y/x) \vdash G}{H, \exists x.F(x) \vdash G}\, \exists_g(*) \qquad \frac{H, F(t/x) \vdash G}{H, \forall x.F(x) \vdash G}\, \forall_g$$

$(*)$ y est non libre dans H et G

$$\frac{H, A, B \vdash G}{H, A \wedge B \vdash G}\, \wedge_g \qquad \frac{H, A \vdash G \quad H, B \vdash G}{H, A \vee B \vdash G}\, \vee_g$$

Elle se fait par induction sur la méta-règle :
- $\mathcal{I}(H, let(x, meta))\sigma = \mathcal{I}(H, meta)\sigma$, d'où l'on déduit par induction $(H, \phi_i(meta) \vdash G)\sigma$. Si $\phi_i(meta) = \top$, alors le résultat est clair. Sinon, la variable x n'étant pas une méta-variable, elle n'est pas substituée. Elle ne peut pas apparaître dans l'image d'une méta-variable libre, car celles-ci ont été introduites avant x. Comme la variable x n'apparaît pas ailleurs on a donc

  $$\frac{(H, \phi_i(meta) \vdash G)\sigma}{(H, \exists x.\phi_i(meta) \vdash G)\sigma}\, \exists_g$$

  Or $\exists x.\phi_i(meta) = \phi_i(let(x, meta))$.
- $\mathcal{I}(H, search(x, meta))\sigma = \mathcal{I}(H, meta)\sigma$, d'où l'on déduit par induction $(H, \phi_i(meta) \vdash G)\sigma$. Si $\phi_i(meta) = \top$, alors le résultat est clair. Sinon, la variable x est une méta-variable. Si elle appartient au domaine de σ, alors $\sigma(x)$ est un terme n'utilisant pas les variables introduites avant et x n'apparaît pas dans l'image de σ, donc x n'est pas libre dans $(H, \phi_i(meta) \vdash G)\sigma$. On peut alors remplacer le terme $\sigma(x)$ par x et lier par un quantificateur existentiel :

  $$\frac{(H, \phi_i(meta) \vdash G)\sigma}{(H, \forall x.\phi_i(meta) \vdash G)\sigma}\, \forall_g$$

  Or $\forall x.\phi_i(meta) = \phi_i(search(x, meta))$.
- $\mathcal{I}(H, assume(F, meta))\sigma = \mathcal{I}(F :: H, meta)\sigma$, d'où l'on déduit par induction $(F, H, \phi_i(meta) \vdash G)\sigma$. Si $\phi_i(meta) = \top$, alors on peut l'éliminer, ce qui donne le résulat. Sinon, par utilisation de la règle $\wedge_g$ on obtient $(H, F \wedge \phi_i(meta) \vdash g)\sigma$. Or $F \wedge \phi_i(meta) = \phi_i(assume(F, meta))$.

– $\mathcal{I}(H, proof)\sigma = (H \vdash G)\sigma$. Or $\phi_i(proof) = \top$, donc c'est terminé.
– $\mathcal{I}(H, then(meta_1, meta_2))\sigma = \mathcal{I}(H, meta_1)\sigma \cup \mathcal{I}(H, meta_2)\sigma$. Des deux ensembles on déduit par induction $(H, \phi_i(meta_1) \vdash G)\sigma$ et $(H, \phi_i(meta_2) \vdash G)\sigma$. Si l'une ou les deux formules sont la formule $\top$, alors c'est terminé. Sinon grâce à la règle $\vee_g$ nous obtenons $(H, \phi_i(meta_1) \vee \phi_i(meta_2) \vdash G)\sigma$. Or $\phi_i(then(meta_1, meta2)) = \phi_i(meta_1) \vee \phi_i(meta_2)$. □

2.3.3 Validation d'une commande

- L'arbre de preuve est parcouru depuis la racine vers les feuilles. Chaque règle est validée selon les cas :
  – **valide** : il n'y a rien à faire.
  – **trivial** : le but courant doit être prouvé.
  – **déduit** : les formules supplémentaires doivent être prouvées l'une après l'autre dans le contexte courant, indépendamment. Il est à noter que l'interprétation de la phrase *deduce F deduce G* est différente de celle de *deduce F and G*. En effet dans le premier cas on déduit F de l'ensemble des hypothèses courantes puis après avoir ajouté F à ces hypothèses on déduit G, alors que dans l'autre cas, on déduit F et on déduit G avec l'ensemble des hypothèses courantes, sans rien ajouter pour G. Il est donc possible de faire des déductions en cascades ou indépendantes.
  – **prouve** : le but qui justifie la règle doit être prouvé.

  Note 2.3.23 les propositions 2.3.16 et 2.3.22 permettent de justifier le fait que l'on peut substituer toute méta-variable par un terme ne contenant que des variables introduites précédemment. C'est à dire que la règle est valide quelle que soit la valeur que l'on souhaite donner aux méta-variables respectant cette condition sur les variables.
  Il est entendu que lors de la justification d'une règle pour laquelle le but courant possède des méta-variables, ces méta-variables sont considérées comme des constantes et ne peuvent être instanciées. Nous verrons cependant dans le chapitre 4 à la section 4.4.4 qu'il pourrait être possible de faire autrement.

- Si dans les indices donnés durant la description des règles avec les commandes BY [...] WITH il y a des formules données il y a deux cas :
  – Si la formule fait partie des hypothèses alors rien de plus n'est fait que de donner un poids faible au démonstrateur à cette hypothèse. Nous reviendrons sur cette notion de poids dans le chapitre 4.
  – Si la formule ne fait pas partie des hypothèses, alors le démonstrateur doit prouver cette formule avec les hypothèses courantes. S'il y parvient, alors cette formule est ajoutée momentanément aux hypothèses courantes avec un poids faible, et disparait ensuite des buts de l'arbre de preuve, comme décrit par les preuves. Cela a un grand avantage car il n'y a jamais explicitement d'élimination d'hypothèse, comme nous l'avons déjà signalé, dans les preuves en langue naturelle. Par contre on peut utiliser très localement une formule qui justifie une étape de démonstration.
- Si après un BY H est donné un WITH avec une égalité $x = t$, alors le démonstrateur doit pouvoir appliquer prioritairement l'hypothèse H avec le terme t à la place de x si H est de la forme $\forall x.K(x)$.
- Si après BY H est donné un WITH K alors l'utilisation de K par H doit être favorisée par le démonstrateur.

Certaines règles peuvent éventuellement être validées autrement. Nous avons parlé dans le chapitre 1 à la section 1.3.2 de la difficulté de voir que la formule but courante peut avoir changé. On peut penser qu'il y a de fortes chances que ce soit le cas lorsque l'on a une méta-règle avec une seule prémisse et où le but n'est pas changé.
Prenons par exemple le cas où l'on a un théorème de la forme $H \to K$ à prouver. Parfois l'écriture de la preuve se fait sous la forme "supposons H". Implicitement on sait qu'il suffit alors de prouver K. Si l'on traduit sans se poser de question cette phrase dans le langage

restreint, on obtient alors "assume H". Cette commande décrit une méta-règle avec une seule prémisse, avec une hypothèse supplémentaire, et pas de nouveau but.
Le raisonnement reste vrai pour les preuves des théorèmes de la forme $\forall \epsilon > 0, P(\epsilon)$, qui généralement commencent par "Soit ϵ positif". Il faut bien se rappeler que $\forall \epsilon > 0, P(\epsilon)$ signifie $\forall \epsilon . \epsilon > 0 \rightarrow P(\epsilon)$, et donc il s'agit bien ici de règles d'introduction.
Dans de tels cas on doit pouvoir détecter l'utilisation de règles d'introduction, qui ajoutent des hypothèses et changeant le but courant. Les difficultés peuvent survenir lorsqu'il s'agit de formules assez complexes, car alors il devient nécessaire de chercher à unifier les formules introduites dans la commande avec des sous-formules de la formule but, l'ordre donné pouvant être différent dans la commande et dans la formule.
On a cependant également ce genre de phénomène, où l'on ajoute des hypothèses sans changer de but, sans pour autant faire de règle d'introduction. Par exemple lorsque l'on a une hypothèse de la forme $\exists x . P(x)$, on écrit alors "soit x tel que $P(x)$".
On préfèrera alors l'interprétation naturelle dans le système, qui considère que l'on est dans le dernier cas, à l'autre qui considère que l'on est dans un cas d'utilisation de règles d'introduction cachée. On peut imaginer que la seconde interprétation peut être prise en compte seulement si la première échoue. Dans l'état actuel du système la seconde interprétation n'est pas autorisée, et il faut absolument donner le nouveau but si celui-ci est changé.

Chapitre 3

Un λ-calcul typé avec deux flèches

Dans le cadre des grammaires catégorielles abstraites (ACG) introduites par P. de Groote (voir [DeG]), il est possible de faire des traductions entre divers formalismes, par exemple entre une structure syntaxique et une structure sémantique, le tout dans un cadre général homogène. Développé à la base sur le lambda calcul linéaire utilisé en linguistique, le système connaît des limites dans l'expressivité notamment dans le cadre sémantique, où l'on souhaite pouvoir utiliser plusieurs fois une même variable.
L'utilité de la linéarité étant importante pour l'analyse syntaxique de textes en langue naturelle, on souhaite conserver son usage. Mais on souhaite également pouvoir exprimer simplement des formules du type (All $\lambda x.$(AND $(A\ x)(B\ x)$)), signifiant $\forall x.A(x) \wedge B(x)$, que l'on peut trouver dans une sémantique, et utilisent de la non linéarité.
La volonté de conserver le cadre homogène des ACGs, permettant une composition naturelle entre elles, a alors mené à l'idée d'introduire dans le lambda calcul linéaire de l'intuitionnisme. Cela fait donc un λ-calcul avec deux types de variables et surtout deux types de flèches (intuitionniste et linéaire). Nous définirons ce calcul dans la première section.
Il est alors naturel de se demander quel peut être le type principal d'un terme écrit dans ce λ-calcul et quelles sont ses propriétés. L'algorithme de typage principal sera donné dans la seconde section.
Une difficulté vient du fait que lors de la recherche du type d'un terme, des flèches peuvent être sous-spécifiées ($-?$), c'est à dire qu'elles peuvent être indifféremment remplacées soit par une flèche intuitionniste ($\rightarrow$) soit par une flèche linéaire ($\multimap$).
Afin de ne pas compliquer ce λ-calcul, il est préférable de ne pas avoir à garder les flèches sous-spécifiées et donc d'être capable de donner des cas pour lesquels on peut avoir une notion de type principal sans ces flèches.
Dans le cas général, nous verrons que c'est impossible. Mais il existe deux cas, le cas linéaire et le cas η-long, pour lesquels nous avons un résultat. Nous traiterons ces deux calculs dans les sections suivantes.
Nous reviendrons dans une dernière section aux ACGs. En effet, la recherche du type principal d'un terme est une des étapes nécessaires à leur utilisation. Nous expliquerons donc le lien entre type principal et ACGs. Nous verrons également que les termes η-longs se retrouvent dans les ACGs.

3.1 Le système de typage

3.1.1 Types

Donnons ici les grammaires définissant les types :
Donnons-nous un ensemble $\mathcal{A}$ de types (constants) atomiques. Les types sont alors définis par :

$$\beta ::= a \mid \alpha \mid \beta \multimap \beta \mid \beta \rightarrow \beta$$

Où a est dans $\mathcal{A}$ et α est une variable de type (appelée aussi variable atomique). La flèche $\multimap$ désigne la flèche linéaire et $\to$ la flèche intuitionniste.
A cette grammaire, nous ajoutons des variables de flèches. Nous noterons $-?$ ou bien $-?_i$ une telle flèche, que nous appelons flèche sous-spécifiée, car elle peut être remplacées par une quelconque des deux flèches. Nous ne notons pas ces flèches dans la grammaire car elles sont un outil pour le typage des termes, et notre but ici est de donner des cas permettant de les faire disparaître.
Nous noterons $\rightsquigarrow$ une flèche quelconque, c'est à dire linéaire, intuitioniste ou sous-spécifiée. Elle nous servira à simplifier les cas dans les définitions et preuves faites par induction sur la complexité des types.

Définition 3.1.1 *Nous définissons ce que signifie être positif ou négatif pour un sous-type, par induction sur la complexité des types. Soit T un type et T' un sous-type de T alors :*
- *Si T'=T alors T' est positif dans T ;*
- *Si $T = T_1 \rightsquigarrow T_2$ et T' est un sous-type de T_1 alors T' est négatif dans T si T' est positif dans T_1, sinon T' est positif dans T ;*
- *Si $T = T_1 \rightsquigarrow T_2$ et T' est un sous-type de T_2 alors T' est positif dans T si T' est positif dans T_2, sinon T' est négatif dans T.*

Nous utiliserons également le terme "polarité" pour désigner la positivité ou la négativité d'un sous-type dans un type.

Définition 3.1.2 *Soit T un type, nous définissons la tête de T par :*
- *Si $T = a$ alors la tête de T est a.*
- *Si $T = A \rightsquigarrow B$ alors la tête de T est $\rightsquigarrow$.*

Définition 3.1.3 *Soit T un type et soit $\rightsquigarrow$ une flèche quelconque apparaissant dans T. La flèche $\rightsquigarrow$ apparaît comme la tête d'un sous-type T' de T. Si T' est négatif pour T alors $\rightsquigarrow$ est négative pour T, sinon $\rightsquigarrow$ est positive pour T.*

Définition 3.1.4 *Soit T un type et a un atome (resp. α une variable de type, resp. $\rightsquigarrow$ une flèche) de T on dit que a (resp. α, resp. $\rightsquigarrow$) est unique dans T s'il apparaît une unique fois dans T.*

3.1.2 Substitution

Définition 3.1.5 *Nous appelons substitution une application S des variables de type dans les types et des flèches sous-spécifiées dans les flèches. Nous définissons alors $S(T)$ (noté aussi ST) l'image d'un type T par une substitution S par induction sur le type :*
- *$S(a) = a$;*
- *$S(\alpha) = S(\alpha)$ défini par S ;*
- *$S(T_1 \multimap T_2) = S(T_1) \multimap S(T_2)$;*
- *$S(T_1 \to T_2) = S(T_1) \to S(T_2)$;*
- *$S(T_1 -?_1 T_2) = S(T_1)\ S(-?_1)\ S(T_2)$.*

Notons que les flèches $\to$ et $\multimap$ ne sont pas modifiées par la substitution. Nous pouvons écrire $S \multimap = \multimap$ et $S \to = \to$.

Lemme 3.1.6 *Soit T un type et S une substitution. Soit T' un sous-type de T. Alors la polarité de ST' dans ST est la même que celle de T' dans T.*

Preuve : par induction sur T
- $T = T'$: c'est clair.
- $T = T_1 \rightsquigarrow T_2$ et T' est un sous-type de T_1 : $ST = ST_1S \rightsquigarrow ST_2$ et par induction la polarité de T' dans T_1 est la même que celle de ST' dans ST_1. Or T_1 a même polarité dans T que ST_1 dans ST. D'où le résultat.
- $T = T_1 \rightsquigarrow T_2$ et T' est un sous-type de T_2 : idem. □

Définition 3.1.7 *Soit S une substitution. Nous dirons que S est une substitution SL si son domaine ne contient que des flèches sous-spécifiées, et qu'elle transforme ces flèches sous-spécifiées en $\multimap$.*

3.1.3 Unification

Etant donnés deux types T_1 et T_2, on cherche à savoir s'ils sont unifiables, c'est à dire s'il existe une substitution S telle que $ST_1 = ST_2$. Il s'agit d'unification du premier ordre. On peut en effet voir un type comme un terme écrit à l'aide d'une seule fonction $fleche$ d'arité trois (i.e. ayant trois arguments) et de constantes. Par exemple, le type

$$(a \to \alpha)\text{–}?_1(b \multimap \beta)\text{–}?_2 c$$

peut être vu comme

$$fleche(x_1, fleche(int, a, \alpha), fleche(x_2, fleche(lin, b, \beta), c))$$

Il y a donc la notion d'unificateur le plus général comme dans tout problème d'unification du premier ordre. Nous noterons $U(A, B)$ l'unificateur le plus général de deux types A et B.

3.1.4 Termes

Voici la grammaire pour les termes :

$$t ::= c \mid x \mid \not\lambda x.t \mid \lambda x.t \mid (t)\ t$$

où c est une constante, x est une variable. Le $\not\lambda$ désigne l'abstraction linéaire, le λ désigne l'abstraction intuitionniste. Nous noterons également $\not\lambda^*$ une abstraction quelconque. On pose des restrictions sur les règles linéaires, à savoir que l'on ne peut former le terme $\not\lambda x.t$ que lorsque x apparaît libre exactement une fois dans t. Le terme $(t_1)\ t_2$ ne peut quant à lui être formé que lorsque les variables linéaires libres de t_1 et t_2 sont disjointes.

Notations : nous écrirons $(t_1)\ t_2\ t_3\ t_4$ le terme $(((t_1)\ t_2)\ t_3)\ t_4$ et $\lambda xy.t$ le terme $\lambda x.\lambda y.t$.

Remarque 3.1.8 Il existe deux types d'abstraction mais un seul type d'application. il aurait certes été possible de définir un calcul avec une application intuitionniste et une application linéaire. Un calcul de cette nature a été défini dans [CePfe1]. Il offre une simplification pour le typage principal qui ne pose alors plus de difficulté. Mais il demande à l'utilisateur de préciser le type des applications, ce qui peut être fastidieux. Il est alors préférable pour l'utilisateur de donner la difficulté à la machine et non à lui-même. C'est la raison pour laquelle il n'y a qu'une seule application.

3.1.5 Règles de typage

Nous définissons maintenant les règles de typage.
Soit C un ensemble de constantes de termes. Soit τ une application de C dans les types qui associe à chaque constante son type (sans flèche sous-spécifiée).
La notation $[\Gamma\ ;\ \Delta] \vdash t : \beta$ désigne un jugement de typage, où
– Γ est un ensemble de déclarations de variables intuitionnistes libres $x : \gamma$;
– Δ est un ensemble de déclarations de variables linéaires libres $x : \gamma$;
– Γ et Δ sont disjoints ;
– t est un terme ;
– β est le type de t.
Nous donnons ci-dessous les règles permettant de typer les termes :

$$\frac{}{[\Gamma\ ;\ ] \vdash c : \tau(c)}$$

$$\frac{}{[\Gamma\ ;\ x\ :\ \gamma]\ \vdash\ x\ :\ \gamma} \qquad \frac{}{[\Gamma, x\ :\ \gamma\ ;\ ]\ \vdash\ x\ :\ \gamma}$$

$$\frac{[\Gamma\ ;\ \Delta, x\ :\ \alpha]\ \vdash\ t\ :\ \beta}{[\Gamma\ ;\ \Delta]\ \vdash\ \text{Ɂ}x.t\ :\ \alpha \multimap \beta} \qquad \frac{[\Gamma, x\ :\ \alpha\ ;\ \Delta]\ \vdash\ t\ :\ \beta}{[\Gamma\ ;\ \Delta]\ \vdash\ \lambda x.t\ :\ \alpha \to \beta}$$

$$\frac{[\Gamma\ ;\ \Delta_1]\ \vdash\ t\ :\ \alpha \multimap \beta \quad [\Gamma\ ;\ \Delta_2]\ \vdash\ u\ :\ \alpha}{[\Gamma\ ;\ \Delta_1, \Delta_2]\ \vdash\ (t)\ u\ :\ \beta}\ (*) \qquad \frac{[\Gamma\ ;\ \Delta]\ \vdash\ t\ :\ \alpha \to \beta \quad [\Gamma\ ;\ ]\ \vdash\ u\ :\ \alpha}{[\Gamma\ ;\ \Delta]\ \vdash\ (t)\ u\ :\ \beta}$$

$(*)$ $Dom(\Delta_1) \cap Dom(\Delta_2) = \emptyset$

Donnons quelques explications sur ces règles. Les règles sur la colonne de gauche sont les règles linéaires, celles de la colonne de droite sont les règles intuitionnistes.
Pour l'introduction des variables le contexte intuitionniste est toujours quelconque, cependant comme il faut respecter le fait que les variables linéaires apparaissent une unique fois, le contexte linéaire doit être vide lors de l'introduction d'une variable intuitionniste et ne comporter que la variable introduite dans le cas linéaire.
Pour l'introduction des flèches, les règles sont assez simples, il n'y a pas de condition particulière.
Pour l'élimination des flèches, le contexte intuitionniste est quelconque, mais identique dans chacune des prémisses (d'où le contexte quelconque lors de l'introduction des variables). Pour la règle linéaire, la condition sur les contextes linéaires est standard, puisqu'il faut conserver l'unicité des variables linéaires dans le terme. Pour la règle intuitionniste, on remarque que le contexte linéaire de la prémisse de droite doit être vide. La raison en est que si t est de la forme $\lambda x.t'$, le terme $(t)u$ est alors un redex. Lors de la réduction à $t[x := u]$, il faut être certain qu'aucune variable linéaire libre de u n'est dupliquée. Or la variable x peut apparaître un nombre arbitraire de fois. Par conséquent u ne doit pas contenir de variables linéaires libres.

Par la suite nous ferons un abus de langage qui consiste à dire qu'un terme est typé si l'on peut le déduire des règles de typage (s'il est dérivable à partir des règles). C'est à dire qu'un terme est typable s'il possède un arbre de typage.
Parfois des jugements serons notés $x\ :\ \tau\ \vdash\ t\ :\ \theta$. Cela signifie qu'une seule variable est considérée, même s'il y en a plusieurs et le fait qu'elle soit linéaire ou intuitionniste n'a pas d'importance.
Notons que dans ce système, aucune flèche sous-spécifiée n'apparaît.

Elargissons les définitions de polarité de sous-types et de flèches pour les jugements.

Définition 3.1.9 *Considérons un jugement $x\ :\ \tau\ \vdash\ t\ :\ \theta$. Alors un sous-type/flèche positif (resp. négatif) de τ est négatif (resp. positif) dans le jugement. Un sous-type/flèche positif (resp. négatif) de θ est positif (resp. négatif) dans le jugement.*

Remarque 3.1.10 Nous avons également une notion de β-réduction, similaire à celle du λ-calcul standard. On retrouve certains résultats du λ-calcul simplement typé comme la forte normalisation par exemple. Pour cela il suffit de traduire ces termes et leur type par les termes et types standards. À une réduction d'un terme correspond exactement une réduction dans sa traduction.

Voici quelques lemmes fort simples mais nécessaires. Les preuves, très faciles, seront peu décrites.

Lemme 3.1.11 *Les termes construits à partir des règles de jugement de bon typage vérifient les restrictions faites sur les termes linéaires. Plus précisément, supposons que l'on ait $[\Gamma\ ;\ \Delta]\ \vdash\ t\ :\ \beta$. Alors*
– si $t = \text{Ɂ}x.t'$ alors x apparaît libre exactement une fois dans t' ;
– si Δ contient une variable x alors x apparaît libre exactement une fois dans t ;
– si $t = (t_1)\ t_2$ alors les variables linéaires libres de t_1 et t_2 sont distinctes.

Preuve : par induction sur la taille de l'arbre.
- Dans le cas des règles d'introduction des variables et de constante, c'est clair.
- Dans le cas de l'abstraction par une variable intuitionniste, c'est clair par induction.
- Dans le cas de l'abstraction par une variable linéaire x, par induction x est libre dans t' et apparaît exactement une fois. Les variables libres de t étant celles de t' sauf x, on a le résultat.
- Dans le cas des applications, ce sont l'induction et les conditions données sur les règles qui permettent de conclure. □

Avant un autre lemme, donnons une définition :

Définition 3.1.12 *Appliquer une substitution S à un jugement, c'est appliquer S à chacun des types du jugement. Appliquer une substitution S à un arbre c'est appliquer S à chacun des jugements qui compose l'arbre.*

Lemme 3.1.13 *Soit $\mathcal{T}$ un arbre de typage pour un terme t. Soit S une substitution. Alors l'image par S de l'arbre de typage est un arbre de typage pour t.*

Preuve : par induction sur la taille de l'arbre
- Dans les cas d'introduction des variables et de constante c'est clair.
- Dans les autres cas, c'est l'induction qui permet de conclure, plus le fait que les substitutions ne changent pas les flèches $\multimap$ et $\to$. □

3.2 Algorithme de typage

Nous allons reprendre très largement l'algorithme de typage de Damas et Milner (cf. [DaMi]). Mais il y a des aménagements à faire à cause de la présence des deux flèches. On peut en effet remarquer que quand on cherche à typer une application $(t_1)\ t_2$, si t_2 n'a pas de variable linéaire libre, il est possible que t_1 puisse avoir soit un type $\multimap$, soit un type $\to$. Il est en effet important de noter que l'application est indifférenciée, c'est à dire qu'elle est la même pour le cas linéaire et le cas intuitionniste. Par conséquent, lors de l'algorithme de typage, il va falloir introduire des flèches $-?_n$ sous-spécifiées. Certaines pourrons alors être identifiées par la suite, mais d'autres vont rester sous-spécifiées.

3.2.1 Un schéma de typage

Le système de typage est augmenté du schéma suivant :

$$\frac{[\Gamma\ ;\ \Delta] \vdash t : \alpha -?_n \beta \quad [\Gamma\ ;\ ] \vdash u : \alpha}{[\Gamma\ ;\ \Delta] \vdash (t)\ u : \beta}$$

Nous avons un lemme semblable à ceux vus plus haut :

Lemme 3.2.1 *Soit t un terme typé dans le système augmenté du schéma ci-dessus. Soit S une substitution. Alors l'image par S de l'arbre de typage est un arbre de typage dérivable.*

Preuve : Elle est la même que pour le lemme sans le schéma. Il faut en plus remarquer pour le schéma que si la flèche est modifiée par la substitution, la règle reste valide dans tous les cas, les fèches sous-spécifiées ayant été introduites pour cette raison précise. □

Et voici un corollaire immédiat :

Lemme 3.2.2 *Soit t un terme typé dans le système avec le schéma. Soit S une substitution de domaine les flèches sous-spécifiées de l'arbre de typage et d'image $\{\to, \multimap\}$. Alors l'image par S de l'arbre de typage est un arbre de typage pour le système initial.*

Preuve : Le fait que ce soit un arbre de typage est conséquence immédiate du lemme précédent et comme le schéma n'est plus utilisé puisque les flèches sous-spécifiées ont disparu, c'est un arbre de typage pour le système initial. □

3.2.2 Type principal d'un terme

Définition 3.2.3 *L'égalité de deux types est définie à renommage des variables de type près.*

Cela revient à dire que deux types T_1 et T_2 sont égaux si il existe S_1 et S_2 deux substitutions telles que $S_1T_1 = T_2$ et $S_2T_2 = T_1$. Nous noterons simplement $T_1 = T_2$.

Définition 3.2.4 *Soit t un terme. Un type T pour t est principal si pour tout T' type de t, il existe une substitution S telle que $ST = T'$.*

Il est clair qu'un terme ayant un type principal a un unique type principal, nous le noterons $\mathrm{TP}(t)$.

Définition 3.2.5 *Un arbre de typage principal d'un terme t est un arbre $\mathcal{T}$ tel que pour tout arbre de typage $\mathcal{T}'$ de t il existe une substitution S telle que $S\mathcal{T}=\mathcal{T}'$.*

Il est clair que le type de t donné par un tel arbre est principal.

Lemme 3.2.6 *Soit t et t' deux termes ayant un type principal. Nous supposons que si t est typable de type T alors t' est aussi typable de type T et réciproquement. Alors t et t' ont même type principal.*

Preuve : le type $\mathrm{TP}(t)$ est un type pour t donc c'est aussi un type pour t'. Par conséquent il existe S' telle que $S'\mathrm{TP}(t')=\mathrm{TP}(t)$. De même il existe S telle que $S\mathrm{TP}(t)=\mathrm{TP}(t')$. Donc $\mathrm{TP}(t)=\mathrm{TP}(t')$. □

3.2.3 Les algorithmes de typage

Algorithme 1

Voici un algorithme de typage pour les termes, qui à un terme dont on connaît toutes les variables libres et en particulier leur genre (linéaire ou intuitionniste) donne un arbre de typage si le terme est typable.

Entrée : un terme t dont les variables linéaires et intuitionnistes sont connues.
Sortie : un arbre de typage $\mathcal{T}$ de t si l'algorithme termine avec succès.

Algorithme : Par induction sur la forme de t.
– Si t=x, t n'a que x comme variable libre. Soit α une variable de type alors $\mathcal{T}$ est

$$\frac{}{[\ ;\ x\ :\ \alpha]\ \vdash\ x\ :\ \alpha}$$

si x est linéaire,

$$\frac{}{[x\ :\ \alpha\ ;\ ]\ \vdash\ x\ :\ \alpha}$$

si x est intuitionniste.
– Si t=c alors t n'a pas de variables libres et $\mathcal{T}$ est

$$\frac{}{[\ ;\ ]\ \vdash\ c\ :\ \tau(c)}$$

– Si $t = \lambda x.t'$, soit $\mathcal{T}'$ l'arbre de typage de t' s'il existe. Soit $\mathcal{T}''$ défini comme étant $\mathcal{T}'$ si x est libre dans t' et étant $\mathcal{T}'$ où l'on ajoute $x\ :\ \alpha$ du coté intuitionniste de tous les jugements de $\mathcal{T}$ avec α une nouvelle variable de type sinon. Si la racine de $\mathcal{T}''$ est $[\Gamma, x\ :\ \alpha\ ;\ \Delta]\ \vdash\ t'\ :\ \beta$ Alors $\mathcal{T}$ est

$$\frac{\mathcal{T}''}{[\Gamma\ ;\ \Delta]\ \vdash\ \lambda x.t'\ :\ \alpha \to \beta}$$

– Si $t = \lambda\!\!\!^{\circ} x.t'$, soit $\mathcal{T}'$ l'arbre de typage de t' s'il existe.
La variable x étant libre dans t' (elle est linéaire), la racine de $\mathcal{T}'$ est de la forme $[\Gamma\ ;\ \Delta, x\ :\ \alpha] \vdash t'\ :\ \beta$ et $\mathcal{T}$ est

$$\frac{\mathcal{T}'}{[\Gamma\ ;\ \Delta] \vdash \lambda\!\!\!^{\circ} x.t'\ :\ \alpha \multimap \beta}$$

– Si $t = (t_1)\ t_2$, soit $\mathcal{T}_1$ et $\mathcal{T}_2$ les arbres de typage pour t_1 et t_2. Nous supposons que les variables de type sont distinctes dans $\mathcal{T}_1$ et $\mathcal{T}_2$ (on peut toujours faire un renommage en appliquant une substitution dans le cas contraire). Notons $[\Gamma_1, \Gamma'_1\ ;\ \Sigma_1] \vdash t_1\ :\ \tau$ le jugement de t_1 et $[\Gamma_2, \Gamma'_2\ ;\ \Sigma_2] \vdash t_2\ :\ \alpha$ celui de t_2 où Γ_1 et Γ_2 regroupent les variables communes de t_1 et t_2 et où Γ'_1 et Γ'_2 sont les variables libres qui n'apparaissent pas dans l'autre terme.
Soit S l'unificateur le plus général des types des variables libres communes de t_1 et t_2. Alors $S\Gamma_1 = S\Gamma_2$. Nous notons maintenant $\mathcal{T}'_1$ l'arbre $\mathcal{T}_1$ dans lequel on a ajouté les variables Γ'_2 manquantes de t_2 et $\mathcal{T}'_2$ l'arbre $\mathcal{T}_2$ dans lequel on a ajouté les variables Γ'_1 manquantes de t_1.
Soit β une nouvelle variable de type. On unifie alors $S\tau$ et $S\alpha \rightsquigarrow_1 \beta$ où $\rightsquigarrow_1$ est $\multimap$ si t_2 a des variables libres linéaires (i.e. si Δ_2 est non vide) et est une nouvelle flèche sous-spécifiée sinon. Nous obtenons S' l'unificateur le plus général et $\mathcal{T}$ est

$$\frac{S'S\mathcal{T}'_1 \qquad S'S\mathcal{T}'_2}{S'S([\Gamma_1, \Gamma'_1, \Gamma'_2\ ;\ \Sigma_1, \Sigma_2] \vdash (t_1)\ t_2\ :\ \beta)}$$

□

Algorithme 2

Il est possible de définir un deuxième algorithme qui lui est équivalent. Nous utiliserons plus tard l'un ou l'autre des algorithmes, selon ce qui conviendra le mieux aux preuves. On considère dans cet autre algorithme le plus d'applications possibles et ainsi les termes sont pris de la forme
– $(x)t_1 \dots t_n\ n \geq 0$;
– $(c)t_1 \dots t_n\ n \geq 0$;
– $\lambda x.u$;
– $\lambda\!\!\!^{\circ} x.u$;
– $(u)t_1 \dots t_n$ avec u une abstraction (de la forme $\lambda\!\!\!^{*} x.u'$), $n \geq 1$.
L'algorithme est le même que le précédent pour les abstraction. Pour l'application d'un terme t (qui peut être une variable, une constante ou une abstraction) à n arguments $t_1, \dots, t_n$, celui-ci est plus complexe. Il y a cependant le même type d'étapes successives :
– Par induction le terme et les arguments sont typés (pour une variable on donne une nouvelle variable de type) ;
– Le type des variables libres communes est unifié ;
– Le type de t obtenu est unifié au type construit à partir de celui des t_i. C'est à dire que l'on prend une nouvelle variable de type β, et si l'on note T_i les types des t_i, alors le type construit est $T_1 \rightsquigarrow_1 \dots T_n \rightsquigarrow_n \beta$ où les $\rightsquigarrow_i$ sont $\multimap$ si t_i a des variables linéaires libres, et de nouvelles flèches sous-spécifiées $-\!\!?_i$ sinon.

Note 3.2.7 La partie difficile de ce chapitre n'étant pas les algorithmes de typage mais les propriétés du type principal obtenu, nous préférons ne détailler que le premier algorithme. Nous ne ferons également la preuve du fait que l'on obtient bien le type principal uniquement pour le premier algorithme, qui est plus simple.
Précisons par ailleurs que ces algorithmes ne sont pas les plus efficaces pour l'implémentation et qu'ils ont pour seul but d'être le plus pratique possible pour la suite.

3.2.4 Propriétés

Il nous faut prouver des résultats importants pour la suite.

Lemme 3.2.8 *Soit t un terme pour lequel l'algorithme donne un arbre. Alors l'arbre est un arbre de typage (avec le schéma) pour t.*

Preuve : par induction sur la complexité de t en utilisant le lemme sur l'application d'une substitution à un arbre de typage et en faisant attention aux restrictions de la règle l'élimination de la flèche utilisée pour le typage de l'application.
- Si $t = x$ c'est clair.
- Si $t = c$ c'est clair.
- Si $t = \lambda x.t'$, par induction on a un arbre pour t'. La règle d'introduction de la flèche $\rightarrow$ étant respectée dans l'algorithme, nous obtenons un arbre de typage pour t.
- Si $t = \lambda\!\!\!/\, x.t'$, idem.
- Si $t = (t_1)\ t_2$, par induction nous obtenons des arbres pour t_1 et pour t_2. Nous savons d'après un lemme précédent que nous pouvons appliquer des substitutions tout en conservant le fait d'avoir un arbre de typage.
  Si les unifications terminent avec succès, alors les variables communes dans t_1 et dans t_2 ont le même type, et t_1 a pour type $T_2 \rightsquigarrow \tau$ après substitution, avec $\rightsquigarrow$ une flèche quelconque et T_2 le type de t_2 après substitution.
  Si t_2 a des variables libres linéaires, alors nécessairement $\rightsquigarrow = \multimap$ d'après l'algorithme. Sinon la flèche $\rightsquigarrow$ est quelconque, puisque c'est l'image par la substitution d'une flèche sous-spécifiée.
  Il est dans tous les cas possible d'utiliser la règle d'application pour typer t. □

Proposition 3.2.9 *Soit t un terme typable. Alors l'algorithme appliqué à t termine avec succès et donne un arbre de typage principal.*

Preuve : Par induction sur t.
- Si $t = x$ c'est clair.
- Si $t = c$ c'est clair.
- Si $t = \lambda x.t'$, c'est clair par induction.
- Si $t = \lambda\!\!\!/\, x.t'$, c'est clair par induction (on remarque qu'ici x est nécessairement libre dans t').
- Si $t = (t_1)\ t_2$ alors par induction l'algorithme termine avec succès pour t_1 et t_2 et donne des arbres de typage principaux $\mathcal{T}_1$ et $\mathcal{T}_2$. Ecrivons les types de t_1 et t_2 ainsi :

$$[\Gamma_1, \Gamma_1' \ ; \ \Sigma_1] \vdash t_1 \ : \ \tau$$

$$[\Gamma_2, \Gamma_2' \ ; \ \Sigma_2] \vdash t_2 \ : \ \alpha$$

Nous avons supposé que les variables de type sont distinctes pour $\mathcal{T}_1$ et pour $\mathcal{T}_2$. Comme t est typable, soit $\mathcal{T}$ un arbre de typage de t et regardons la partie qui nous intéresse :

$$\frac{[\Gamma, \Gamma_a, \Gamma_b \ ; \ \Delta_1] \vdash t_1 \ : \ a \rightsquigarrow b \quad [\Gamma, \Gamma_a, \Gamma_b \ ; \ \Delta_2] \vdash t_2 \ : \ a}{[\Gamma, \Gamma_a, \Gamma_b \ ; \ \Delta_1, \Delta_2] \vdash (t_1)\ t_2 \ : \ b}$$

Γ est l'ensemble des variables communes, Γ_a est le reste des variables libres intuitionnistes de t_1 et Γ_b est le reste des variables libres intuitionnistes de t_2. Notons $\mathcal{U}_1$ l'arbre dont la racine est

$$[\Gamma, \Gamma_a, \Gamma_b \ ; \ \Delta_1] \vdash t_1 \ : \ a \rightsquigarrow b$$

et $\mathcal{U}_2$ l'arbre dont la racine est

$$[\Gamma, \Gamma_a, \Gamma_b \ ; \ \Delta_2] \vdash t_2 \ : \ a$$

Nous pouvons supposer que les variables de type qui apparaissent dans $\mathcal{T}$ sont différentes de celles de $\mathcal{T}_1$ et $\mathcal{T}_2$. Comme les arbres obtenus par l'algorithme sont principaux, il existe des substitutions qui égalisent les arbres et donc les jugements. Plus précisément il existe S_1 et S_2 telles que

$$S_1([\Gamma_1, \Gamma_1' \ ; \ \Sigma_1] \vdash t_1 \ : \ \tau) = [\Gamma, \Gamma_a \ ; \ \Delta_1] \vdash t_1 \ : \ a \rightsquigarrow b$$

et

$$S_2([\Gamma_2, \Gamma_2' \ ; \Sigma_2] \vdash t_2 : \alpha) = [\Gamma, \Gamma_b \ ; \Delta_2] \vdash t_2 : a$$

Les variables de type étant distinctes dans $\mathcal{T}_1$, $\mathcal{T}_2$ et $\mathcal{T}$, nous voyons que $S_1 \circ S_2$ unifie Γ_1 et Γ_2, donc les variables intuitionnistes ont des types unifiables. Cela signifie que l'unification des types des variables aboutit, en donnant l'unificateur le plus général S. Il existe alors une substitution R telle que $S_1 \circ S_2 = R \circ S$.
Soit β une nouvelle variable de type et $\rightsquigarrow_1$ la flèche introduite dans l'algorithme. Si $\rightsquigarrow_1$ est linéaire, alors nécessairement $\rightsquigarrow$ l'est également car t_2 contient des variables libres linéaires. Sinon $\rightsquigarrow_1$ est une flèche sous-spécifiée. Nous pouvons considérer alors que la substitution $(\rightsquigarrow_1 \mapsto \rightsquigarrow)$ a un sens, en disant qu'il s'agit de l'identité dans le cas où les flèches sont linéaires. Ainsi $S\tau$ et $S\alpha \rightsquigarrow_1 \beta$ sont unifiables car $(\beta \mapsto b) \circ (\rightsquigarrow_1 \mapsto \rightsquigarrow) \circ R$ est un unificateur.
Nous pouvons par conséquent prendre S' l'unificateur le plus général de $S\tau$ et $S\alpha \rightsquigarrow_1 \beta$, ce qui prouve que l'algorithme termine avec succès. Il reste à prouver que l'arbre de typage est principal. Comme S' est l'unificateur le plus général, il existe une substitution R' telle que $(\beta \mapsto b) \circ (\rightsquigarrow_1 \mapsto \rightsquigarrow) \circ R = R' \circ S'$.
Notons

$$\sigma = (\beta \mapsto b) \circ (\rightsquigarrow_1 \mapsto \rightsquigarrow) \circ S_1 \circ S_2$$

et

$$\tau = (\beta \mapsto b) \circ (\rightsquigarrow_1 \mapsto \rightsquigarrow) \circ R \circ S$$

Nous avons alors $\mathcal{T}$ qui est

$$\begin{aligned}
\mathcal{T} &= \frac{\sigma\mathcal{T}_1' \quad \sigma\mathcal{T}_2'}{\sigma([\Gamma_1, \Gamma_1', \Gamma_2' \ ; \Sigma_1, \Sigma_2] \vdash (t_1)\ t_2 : \beta)} \\
&= \frac{\tau\mathcal{T}_1' \quad \tau\mathcal{T}_2'}{\tau([\Gamma_1, \Gamma_1', \Gamma_2' \ ; \Sigma_1, \Sigma_2] \vdash (t_1)\ t_2 : \beta)} \\
&= \frac{R' \circ S' \circ S\mathcal{T}_1' \qquad R' \circ S' \circ S\mathcal{T}_2'}{R' \circ S' \circ S([\Gamma_1, \Gamma_1', \Gamma_2' \ ; \Sigma_1, \Sigma_2] \vdash (t_1)\ t_2 : \beta)}
\end{aligned}$$

Ainsi l'arbre donné par l'algorithme est principal. □

Remarque 3.2.10 L'algorithme termine toujours, même pour un terme non typable. Dans ce cas, une erreur est signalée lors de l'unification, car l'algorithme d'unification au premier ordre termine toujours, soit par le succès si les termes sont unifiables, soit par un message d'erreur dans le cas contraire.

3.2.5 La propriété SNIP

Notre but dans les prochaines sections est de donner des fragments du calcul dans lesquels il est possible de remplacer toutes les flèches sous-spécifiées en des flèches intuitionnistes. En effet, si l'algorithme de typage donne des flèches sous-spécifiées, il n'est pas souhaitable d'afficher cette information pour l'utilisateur.
Le fait de choisir le remplacement des flèches sous-spécifiées en flèches intuitionnistes provient de ce que l'on peut effectivement trouver un critère de remplacement de certaines flèches intuitionnistes en flèches linéaires tout en gardant un type valide. Les flèches linéaires peuvent elles se trouver partout dans le type d'un terme et donc aucun critère les concernant ne peut être donné.
La notion de type principal change alors. Selon leur polarité dans le type, les flèches intuitionnistes pourrons être remplacées ou non par une flèche linéaire.
Dans chaque fragment où nous allons nous placer, la même propriété sur les flèches sera prouvée sur le type principal donné par l'algorithme. Donnons cette propriété dans une définition :

Définition 3.2.11 (Propriété SNIP) *Soit T un type.*
- *Si T vérifie que ses flèches sous-spécifiées sont négatives et ses flèches intuitionnistes sont positives, nous disons que T a la propriété SNIP.*
- *Si T est tel que les flèches sous-spécifiées sont positives et les flèches intuitionnistes négatives, on dit que T a la propriété SPIN.*
- *Soit $x : \tau \vdash t : \theta$ un jugement de typage. Si les types des variables ont la propriété SPIN et si θ a la propriété SNIP alors le jugement a la propriété SNIP.*

Définition 3.2.12 *Soit T_1 et T_2 deux types. Si T_1 a la propriété SNIP (resp. SPIN) et T_2 a la propriété SPIN (resp. SNIP), nous dirons que T_1 a la propriété opposée à celle de T_2.*

Remarque 3.2.13 Si T a une des deux propriétés SNIP ou SPIN alors $T \multimap \alpha$ a la propriété opposée.

Remarque 3.2.14 Supposons qu'un jugement vérifie la propriété SNIP, et que de plus les flèches sous-spécifiées soient toutes distinctes. Alors les flèches $-?$ sont en position négative et les flèches $\to$ sont en position positive. Comme les flèches sous-spécifiées peuvent être remplacées par n'importe quelle flèche, nous pouvons les remplacer par des $\to$. Dans le nouveau type obtenu, nous savons que chaque flèche intuitionniste en position négative, qui avant était une flèche $-?$, peut être remplacée par une flèche linéaire tout en conservant un type valide, grâce au fait que les flèches sous-spécifiées soient distinctes.
Cela peut nous permettre, pour les types principaux qui vérifient la propriété SNIP avec des flèches sous-spécifiées distinctes, d'obtenir une autre notion de type principal, sans flèche sous-spécifiée, pour laquelle on obtient d'autres types valides en remplaçant toute flèche intuitionniste négative en flèche linéaire.

Note 3.2.15 Toutes les constantes considérées par la suite auront toujours un type déterminé, c'est à dire seulement avec des flèches $\multimap$ et $\to$.

Avant de faire des restrictions sur les termes montrons tout d'abord pourquoi nous en faisons en donnant un exemple.

Exemple 3.2.16 Soit

$$t = \lambda g \lambda f \lambda x \lambda u.(g) \quad (f)\ x \quad (f)\ \lambda t.(t)\ u$$

Son type principal est

$$(b \multimap b -?_1 n) \to (((a -?_2^- e) \to^+ e) \multimap b) \to ((a -?_2^+ e) \to^- e) \multimap a \to n$$

Les symboles $+$ et $-$ en exposant sont seulement des annotations pour ce qui va suivre. Ce type n'a pas la propriété SNIP car aucune des deux conditions n'est remplie :

1. Il y a dans le type une flèche $\to$ qui a une occurrence négative (celle qui a un exposant $-$). La raison est que f prend un argument $\lambda t.(t)\ u$ d'un certain type $((a -?_2 e) \to e)$ et comme x apparaît comme argument de f, x doit avoir le même type. On voit qu'effectivement $((a -?_2 e) \to e)$ apparaît deux fois dans le type, avec deux polarités distinctes.
2. la flèche $-?_2$ a une occurrence positive (celle qui a un exposant $+$) pour la même raison.

- Supposons que l'on remplace $-?_2$ par une flèche intuitionniste. La présence de flèches intuitionnistes dans les deux polarités rend impossible l'existence d'un critère (ou, s'il en existe un, d'un critère simple) portant sur le type et permettant de décider de changer certaines flèches intuitionnistes par des flèches linéaires tout en conservant un type valide pour t. En effet, si dans le type principal une flèche est intuitionniste, elle ne peut pas être remplacée par une flèche linéaire.
- C'est la même chose si l'on suppose que l'on remplace $-?_2$ par une flèche linéaire car il y a des flèches linéaires dans toutes les polarités, ce qui est le cas pour la plupart des termes contenant des variables linéaires.

- Un autre fait important est que la flèche $-?_2$ apparaît deux fois dans le type, et non une unique, dans des polarités différentes. Cela implique que pour conserver un type valide, il faut modifier les deux apparitions par la même flèche, ce qui rend le critère éventuel encore plus complexe que celui que nous proposons. Par la suite nous verrons que dans les fragments étudiés, les flèches sous-spécifiées n'apparaîtrons qu'une seule fois dans le type principal.

On voit alors que t n'est pas en forme η-longue car x n'a pas d'argument bien qu'elle ait un type flèche et t n'est pas non plus linéaire, car la variable f apparaît deux fois.

3.3 Termes linéaires

Nous nous plaçons dans un cadre restreint des termes. Un terme est considéré linéaire si les variables intuitionnistes apparaissent exactement une fois. Autrement dit, il est obtenu en dérivant les règles suivantes de typage :

$$\frac{}{[\,;\,] \vdash c : \tau(c)} \qquad \frac{}{[\,;x : \gamma] \vdash x : \gamma} \qquad \frac{}{[x : \gamma\,;\,] \vdash x : \gamma}$$

$$\frac{[\Gamma\,;\,\Delta, x : \alpha] \vdash t : \beta}{[\Gamma\,;\,\Delta] \vdash \lambda^{\wp} x.t : \alpha \multimap \beta} \qquad \frac{[\Gamma, x : \alpha\,;\,\Delta] \vdash t : \beta}{[\Gamma\,;\,\Delta] \vdash \lambda x.t : \alpha \to \beta}$$

$$\frac{[\Gamma_1\,;\,\Delta_1] \vdash t : \alpha \multimap \beta \quad [\Gamma_2\,;\,\Delta_2] \vdash u : \alpha}{[\Gamma_1, \Gamma_2\,;\,\Delta_1, \Delta_2] \vdash (t)\, u : \beta}\,(*)$$

$$\frac{[\Gamma_1\,;\,\Delta] \vdash t : \alpha \to \beta \quad [\Gamma_2\,;\,] \vdash u : \alpha}{[\Gamma_1, \Gamma_2\,;\,\Delta] \vdash (t)\, u : \beta}\,(*)$$

$(*)$ $Dom(\Delta_1) \cap Dom(\Delta_2) = \emptyset$ et $Dom(\Gamma_1) \cap Dom(\Gamma_2) = \emptyset$

Remarquons qu'un terme typable dans ce système est typable dans le système d'origine. Le système donné ici n'est qu'une restriction permettant de forcer par des règles le typage de termes linéaires uniquement.
L'algorithme de typage est par ailleurs rendu plus simple puisqu'il devient inutile d'unifier le type des variables du fait qu'aucune variable n'est commune à deux termes.

Nous allons demander aux constantes de vérifier la propriété SNIP. C'est à dire que toutes les flèches intuitionnistes sont positives, et toutes les autres flèches sont linéaires (il n'y a pas de flèches sous-spécifiées). Si nous n'avons pas cette condition il est en effet évident que la propriété sur les flèches ne peut pas être conservée car il suffit de prendre pour t une constante dont le type a une flèche intuitionniste négative.

Nous allons donc prouver qu'un terme linéaire a bien la propriété SNIP. Nous prouverons en fait un résultat un peu plus fort, que l'on vérifie très facilement pour le λ-calcul linéaire standard.

Nous avons besoin d'un lemme spécifique aux termes avec constantes.

Lemme 3.3.1 *Soit C un type constant (sans variables de types). Soit T un type unifiable avec C, contenant une variable de type α qui est unique dans T.*
- *Si α est positive (resp. négative) dans T et C vérifie la propriété SNIP alors la substitution obtenue par l'unification transforme la variable de type en un type constant vérifiant la propriété SNIP (resp. SPIN).*
- *Si α est négative (resp. positive) dans T et C vérifie la propriété SPIN alors la substitution obtenue par l'unification transforme α en un type constant vérifiant la propriété SNIP (resp. SPIN).*

Preuve : par induction sur la complexité de C. Notons α la variable de type.

- $T{=}\alpha$: c'est clair (α est le seul atome possible puisqu'il est unique dans T).
- $T = T_1 \to T_2$ avec α dans T_1 : comme T et C sont unifiables et C est constant, $C = C_1 \to C_2$. L'unification commence par unifier T_1 avec C_1. Le type C_1 a la propriété contraire à celle de C. De plus α a la polarité contraire dans T_1 à celle dans T. Cette unification donne une substitution S_1 qui transforme α en un type A constant vérifiant la propriété prévue dans l'énoncé du lemme avec C_1 et T_1. Ensuite on unifie S_1T_2 et $S_1C_2 = C_2$, ce qui donne une substitution S_2. Alors la substitution qui unifie T et C est $S = S_2 \circ S_1$ et ainsi $S(\alpha){=}A$. Comme il faut inverser la polarité de α et la propriété vérifiée par la constante, la propriété est vérifiée.
- $T = T_1 \to T_2$ avec α dans T_2 : comme T et C sont unifiables et C est constant, $C = C_1 \to C_2$. L'unification commence par unifier T_1 avec C_1, qui ont la même propriété. Cette unification donne une substitution S_1. Ensuite on unifie S_1T_2 et $S_1C_2 = C_2$, ce qui donne une substitution S_2. Dans S_1T_2 le type α apparaît et a la même polarité que dans T_2. De plus le type C_2 a la même propriété que C. S_2 transforme alors α en un type constant A qui vérifie la propriété prévue dans l'énoncé du lemme avec C_2 et S_1T_2. Alors la substitution qui unifie T et C est $S = S_2 \circ S_1$ et ainsi $S(\alpha){=}A$. Comme on garde la polarité de α et la propriété vérifiée par la constante, la propriété est vérifiée. □

Intéressons nous maintenant au cas des termes qui sont β-réduits (normaux).

Lemme 3.3.2 *Soit t un terme normal, linéaire et typable. Alors son type principal vérifie les points suivants :*
- *Chaque variable de type qui apparaît n'apparaît que deux fois, avec une occurrence positive et une négative ;*
- *La propriété SNIP est vérifiée ;*
- *Les flèches sous-spécifiées qui apparaissent sont uniques (elles sont toutes distinctes).*

Preuve : par induction sur la complexité du terme. Comme t est normal, il n'y a que quatre cas possibles sur la forme de t :
- $t = \lambda x.t'$: par induction le type principal de t' vérifie les propriétés. Or le type principal de t est le même, hormis le fait que le type de x est placé à droite du séquent, en position négative. On ajoute une flèche $\to$ positive. Donc tout est conservé.
- $t = \parr x.t'$: idem au cas précédent, avec ici l'ajout d'une flèche $\multimap$.
- $t = (x)\ t_1\ \ldots\ t_n$ (où $n \geq 0$) : par induction les t_i vérifient les propriétés. Notons leur type principal $[\Gamma_i\ ;\ \Delta_i] \vdash t_i : \tau_i$. La variable x n'apparaît qu'une fois dans t et les variables des t_i sont disjointes. Selon l'algorithme, on prend une nouvelle variable de type α. La variable x est de type $\tau := \tau_1 \rightsquigarrow_1 \ldots \tau_n \rightsquigarrow_n \alpha$ où $\rightsquigarrow_i$ est $\multimap$ si t_i a des variables linéaires libres et est une nouvelle flèche sous-spécifiée sinon. Le type principal de t est alors $[\{\Gamma_i\}_i, x = \tau\ ;\ \{\Delta_i\}_i] \vdash t : \alpha$ si x est intuitionniste ou $[\{\Gamma_i\}_i\ ;\ \{\Delta_i\}, x = \tau] \vdash t : \alpha$ sinon. Ayant remarqué que les τ_i sont en position négative dans le type de x et que rien n'a disparu on vérifie simplement que les propriétés sont vérifiées : les variables de type n'apparaissent que deux fois, avec une occurrence négative et une positive, la seule ajoutée étant α. Les positions des flèches sont conservées, car seules des flèches sous-spécifiées et linéaires ont été ajoutées et ce en position négative. Enfin, les flèches sous-spécifiées sont uniques, car les seules qui ont été ajoutées sont nouvelles.
- $t = (c)\ t_1\ \ldots\ t_n$ (où $n \geq 0$) : notons $C_1 \rightsquigarrow_{a_1} \ldots C_n \rightsquigarrow_{a_n} C$ le type de c. Les flèches $\rightsquigarrow_{a_i}$ sont soit $\multimap$ soit $\to$, elles ne sont pas sous-spécifiées. Notons également T_i le type de t_i pour tout i. On unifie alors un type fixe, celui de la constante c, et un type T à l'image du cas précédent de la forme $T_1 \rightsquigarrow_1 \ldots T_n \rightsquigarrow_n \alpha$ où chaque $\rightsquigarrow_i$ est soit sous-spécifiée soit linéaire selon t_i et α une nouvelle variable de type. Nous remarquons que C vérifie la propriété SNIP alors que T vérifie la propriété SPIN.
  Après unification, le type de t est C, donc n'a pas de variables de type ni de flèches sous-spécifiées et la propriété SNIP est vérifiée pour t. Il reste à regarder le type des variables libres après substitution. Nous considérons les variables et leur type avant la

substitution dans chaque jugement de type des termes t_i. Les variables de type et flèches sous-spécifiées n'apparaissant que dans des types de variables libres ne sont pas modifiées par la substitution provenant de l'unification puisqu'elles n'apparaissent pas dans le type T. Une flèche sous-spécifiée apparaissant dans le type d'un terme t_i ne se trouve pas dans le type d'une variable libre car elle est unique.
La seule chose qui importe est donc de savoir en quoi sont transformées les variables de type qui apparaissent dans le type d'un terme t_i et dans le type d'une variable libre. C'est à dire en quoi sont transformées les variables de type qui n'apparaissent qu'une fois dans le type $T = T_1 \rightsquigarrow_1 \dots T_n \rightsquigarrow_n \alpha$. Le lemme 3.3.1 nous indique que ces variables sont transformées en des types vérifiant la propriété
– SNIP si elles sont positives ;
– SPIN si elles sont négatives.
Comme ces variables de type apparaissent dans le type des variables dans une polarité opposée à celle dans T, mais comme le type des variables vérifie la propriété SPIN, la substitution appliquée aux types des variable donne bien des types vérifiant la propriété SPIN. □

Nous devons maintenant faire en sorte de pouvoir retrouver la même propriété pour les termes quelconques. Pour cela nous avons besoin de quelques lemmes.

Lemme 3.3.3 *Soit A et B deux types unifiables. Soit S une substitution SL (ne transformant que des flèches sous-spécifiées en $\multimap$). Nous supposons SA et SB unifiables aussi. Alors il existe S' SL telle que*
U(SA,SB)∘S =S′∘ U(A,B).

Preuve : par induction sur la somme des nombres d'étapes d'unification de A et B et de SA et SB. Les deux types A et B étant unifiables, ainsi que SA et SB, nous avons les cas suivants :
– $A = B = c$ (type atomique constant) : S'=S convient.
– $A = a$: U(Sa,SB)=$a \mapsto SB$ car a est laissé invariant par S. La substitution $S' = S$ convient.
– $B = b$: idem.
– $A = A_1 \multimap A_2$ et $B = B_1 \multimap B_2$: par induction soit S'_1 SL telle que U(SA_1,SB_1)∘S =S'_1∘ U(A_1,B_1). Notons T'_1=U(SA_1,SB_1) et T_1=U(A_1,B_1). Alors $T'_1 \circ S = S'_1 \circ T_1$.
Ensuite, U(SA,SB)=$T'_2 \circ T'_1$ où T'_2=U(T'_1SA_2,T'_1SB_2)=U($S'_1T_1A_2$,$S'_2T_1B_2$). Ainsi, en utilisant une deuxième fois l'induction, $T'_2 \circ S'_1$=S'_2∘U(T_1A_2,T_1B_2).
Or U(A,B)=$T_2 \circ T_1$ où T_2=U(T_1A_2,T_1B_2). On en conclut alors l'égalité suivante :
U(SA,SB)∘S=$T'_2 \circ T'_1 \circ S$=$T'_2 \circ S'_1 \circ T_1$=$S'_2 \circ T_2 \circ T_1$=$S'_2$∘ U($A$,$B$).
– $A = A_1 \to A_2$ et $B = B_1 \to B_2$: idem.
– $A = A_1 -?_1 A_2$ et $B = B_1 \multimap B_2$: notons S_1 la substitution qui transforme $-?_1$ en $\multimap$. Traitons les cas :

1. $S(-?_1) = \multimap$: U(SA,SB)∘S=U(SS_1A,SS_1B)∘$S \circ S_1$=S'U(S_1A,S_1B)∘S_1 par induction. En effet, S_1A et S_2A sont unifiés en moins d'étapes que A et B. La première égalité est évidente parce que S contient S_1. Or nous avons également U(A,B)=U(S_1A,S_1B)∘S_1. D'où le résultat,
2. $S(-?_1) = -?_1$: U(SA,SB)∘S=U(S_1SA,S_1SB)∘$S_1 \circ S$=S'U(A,B) par induction. La première égalité est ici par définition.

– $A = A_1 -?_1 A_2$ et $B = B_1 \to B_2$. Nécessairement $S(-?_1) \neq \multimap$, car SA et SB sont unifiables. Donc $S(-?_1) = -?_1$. Le cas se règle comme précédemment.
– $A = A_1 -?_1 A_2$ et $B = B_1 -?_2 B_2$: trois cas (en dehors des cas symétriques)

1. S est l'identité sur les deux flèches : notons S_1 la substitution qui transforme $-?_1$ et $-?_2$ en $\multimap$. Alors U(SA,SB)∘S=U(S_1SA,S_1SB)∘$S_1 \circ S$=S'U(A,B) par induction.
2. S ne transforme que l'une des flèches, $-?_1$ par exemple.

Alors l'unification est U(SA,SB)=U(S_2SA,S_2SB)$\circ S_2$ où S_2 transforme $-?_2$ en $-\circ$. Donc nous avons l'égalité U(SA,SB)$\circ S$=U(S_2SA,S_2SB)$\circ S_2 \circ S$, puis par induction nous obtenons le résultat U(SA,SB)=S'U(A,B).

3. S transforme les deux flèches. Notons S_1 la substitution qui transforme $-?_1$ en $-?_2$. Alors U(SA,SB)$\circ S$=U(SS_1A,SS_1B)$\circ S \circ S_1$=S'U(S_1A,S_1B)$\circ S_1$ par induction. La première égalité vient du fait que S=SS_1. Or U(A,B)=U(S_1A,S_1B)$\circ S_1$. D'où le résultat.

– Les autres cas symétriques, où $B = B_1 \multimap B_2$ et A a une autre flèche : idem aux cas précédents. □

Lemme 3.3.4 *Soit u_1, u_2 et v trois termes typables. On suppose qu'aucune des variables libres de u_1 et de u_2 n'est dans v. Nous supposons qu'il existe S une substitution SL telle que TP(u_1)=S TP(u_2). On a les propositions suivantes :*
– *Si $(u_1)v$ et $(u_2)v$ sont typables alors il existe S' une substitution SL telle que TP($(u_1)v$)=S' TP($(u_2)v$) ;*
– *Si $(v)u_1$ et $(v)u_2$ sont typables alors il existe S' une substitution SL telle que TP($(v)u_1$)=S' TP($(v)u_2$).*

Preuve :
– Grâce à l'hypothèse, définissons A tel que TP(u_1)=S TP(u_2)=S A. Comme les u_i n'ont pas les mêmes variables libres que v, pour typer $(u_1)v$ (resp. $(u_2)v$), il suffit d'unifier S TP(u_2) avec TP(v)$\rightsquigarrow_1 \alpha$ (resp. TP(u_2) avec TP(v)$\rightsquigarrow_1 \alpha$), où α est une nouvelle variable de type et $\rightsquigarrow_1$ est soit une nouvelle flèche sous-spécifiée si v n'a pas de variable libre linéaire, soit $-\circ$. Comme l'algorithme donne des variables de type différentes, S laisse B:=TP(v)$\rightsquigarrow_1 \alpha$ invariant. Donc nous cherchons les unificateurs de SA et SB puis de A et B. Le lemme 3.3 nous affirme que U(SA,SB)$\circ S$=$S'\circ$ U(A,B), où S' est une substitution SL. Alors le type principal de $(u_1)v$ est S' TP($(u_2)v$).
– la preuve est analogue. □

Lemme 3.3.5 *Soit w_1, w_2 et v des termes.*

1. *Supposons que $t = (\lambda x.(w_1)w_2)v$ est typable et que x n'apparaît qu'une seule fois dans $(w_1)w_2$.*
   (a) *Si x est dans w_1 alors TP(t)=TP($((\lambda x.w_1)v)w_2$).*
   (b) *Si x est dans w_2 alors TP(t)=TP((w_1) $(\lambda x.w_2)v$).*
2. *Supposons que $t = (\text{Y} x.(w_1)w_2)v$ est typable et que x n'apparaît qu'une seule fois dans $(w_1)w_2$.*
   (a) *Si x est dans w_1 alors TP(t)=TP($((\text{Y} x.w_1)v)w_2$).*
   (b) *Si x est dans w_2 alors TP(t)=S TP((w_1) $(\text{Y} x.w_2)v$) où S est une substitution SL.*

Preuve : Nous allons prouver ce lemme en utilisant le lemme 3.2.6, qui permet aisément de prouver que deux termes ont le même type principal.

1. la variable x est intuitionniste ici.
   (a) Regardons un premier arbre de typage (dans lequel nous avons omis les variables libres superflues) :

$$\dfrac{\dfrac{\dfrac{[x : \alpha\ ;\ ] \vdash w_1 : \beta \rightsquigarrow_1 \gamma \quad [\ ;\ ] \vdash w_2 : \beta}{[x : \alpha\ ;\ ] \vdash (w_1)w_2 : \gamma}}{[\ ;\ ] \vdash \lambda x.(w_1)w_2 : \alpha \to \gamma} \qquad [\ ;\ ] \vdash v : \alpha}{[\ ;\ ] \vdash (\lambda x.(w_1)w_2)v : \gamma}$$

   et un second :

$$\dfrac{\dfrac{\dfrac{[x : \alpha\ ;\ ] \vdash w_1 : \beta \rightsquigarrow_1 \gamma}{[\ ;\ ] \vdash \lambda x.w_1 : \alpha \to \beta \rightsquigarrow_1 \gamma} \quad [\ ;\ ] \vdash v : \alpha}{[\ ;\ ] \vdash (\lambda x.w_1)v : \beta \rightsquigarrow_1 \gamma} \quad [\ ;\ ] \vdash w_2 : \beta}{[\ ;\ ] \vdash ((\lambda x.w_1)v)w_2 : \gamma}$$

On voit que quand on a typé un des deux termes alors le second est typable du même type. On remarque ici que $\rightsquigarrow_1$ peut être n'importe quelle flèche, puisque les variables libres de l'argument lors de l'élimination de cette flèche sont les mêmes dans les deux cas.

Donc ils ont même type principal, c'est à dire TP(t)=TP($((\lambda x.w_1)v)w_2$).

(b) Regardons l'arbre de typage de t dans ce cas :

$$\dfrac{\dfrac{\dfrac{[\ ;\ ] \vdash w_1 : \beta \rightsquigarrow_1 \gamma \quad [x : \alpha\ ;\ ] \vdash w_2 : \beta}{[x : \alpha\ ;\ ] \vdash (w_1)w_2 : \gamma}}{[\ ;\ ] \vdash \lambda x.(w_1)w_2 : \alpha \to \gamma} \quad [\ ;\ ] \vdash v : \alpha}{[\ ;\ ] \vdash (\lambda x.(w_1)w_2)v : \gamma}$$

Regardons l'arbre de typage de $(w_1)\ (\lambda x.w_2)v$:

$$\dfrac{[\ ;\ ] \vdash w_1 : \beta \rightsquigarrow_1 \gamma \qquad \dfrac{\dfrac{[x : \alpha\ ;\ ] \vdash w_2 : \beta}{[\ ;\ ] \vdash \lambda x.w_2 : \alpha \to \beta} \quad [\ ;\ ] \vdash v : \alpha}{[\ ;\ ] \vdash (\lambda x.w_2)v : \beta}}{[\ ;\ ] \vdash (w_1)(\lambda x.w_2)v : \gamma}$$

Comme v n'a pas de variables linéaires libres (à cause de l'élimination de $\to$ avec v), on voit que quand on a typé un des deux termes alors le second est typable du même type. On remarque ici que $\rightsquigarrow_1$ peut être n'importe quelle flèche, puisque les variables linéaires libres de l'argument lors de l'élimination de cette flèche sont les mêmes. Donc ils ont même type principal, c'est à dire TP(t)=TP($((\lambda x.w_1)v)w_2$).

2. la variable x est linéaire ici.

(a) Regardons l'arbre de typage de t :

$$\dfrac{\dfrac{\dfrac{[\ ;\ x : \alpha] \vdash w_1 : \beta \rightsquigarrow_1 \gamma \quad [\ ;\ ] \vdash w_2 : \beta}{[\ ;\ x : \alpha] \vdash (w_1)w_2 : \beta}}{[\ ;\ ] \vdash \lambda\!\!\!^{\circ} x.(w_1)w_2 : \alpha \multimap \gamma} \quad [\ ;\ ] \vdash v : \alpha}{[\ ;\ ] \vdash (\lambda\!\!\!^{\circ} x.(w_1)w_2)v : \gamma}$$

et celui de $((\lambda\!\!\!^{\circ} x.w_1)v)w_2$:

$$\dfrac{\dfrac{\dfrac{[\ ;\ x : \alpha] \vdash w_1 : \beta \rightsquigarrow_1 \gamma}{[\ ;\ ] \vdash \lambda\!\!\!^{\circ} x.w_1 : \alpha \multimap \beta \rightsquigarrow_1 \gamma} \quad [\ ;\ ] \vdash v : \alpha}{[\ ;\ ] \vdash (\lambda\!\!\!^{\circ} x.w_1)v : \beta \rightsquigarrow_1 \gamma} \quad [\ ;\ ] \vdash w_2 : \beta}{[\ ;\ ] \vdash ((\lambda\!\!\!^{\circ} x.w_1)v)w_2 : \gamma}$$

On voit que quand on a typé un des deux termes alors le second est typable du même type. On remarque ici que $\rightsquigarrow_1$ peut être n'importe quelle flèche, puisque les variables libres de l'argument lors de l'élimination de cette flèche sont les mêmes. Donc ils ont même type principal, c'est à dire TP(t)=TP($((\lambda\!\!\!^{\circ} x.w_1)v)w_2$).

(b) Regardons l'arbre de typage de t dans ce cas :

$$\dfrac{\dfrac{\dfrac{[\,;\,]\vdash w_1 : \beta \multimap \gamma \quad [\,;\, x:\alpha]\vdash w_2 : \beta}{[\,;\, x:\alpha]\vdash (w_1)w_2 : \gamma}}{[\,;\,]\vdash \lambda\!\!\!^{\circ} x.(w_1)w_2 : \alpha \multimap \gamma} \qquad [\,;\, v]\vdash \alpha :}{[\,;\,]\vdash (\lambda\!\!\!^{\circ} x.(w_1)w_2)v : \gamma}$$

Regardons l'arbre de typage de $(w_1)\ (\lambda\!\!\!^{\circ} x.w_2)v$:

$$\dfrac{[\,;\,]\vdash w_1 : \beta \rightsquigarrow_1 \gamma \qquad \dfrac{\dfrac{[\,;\, x:\alpha]\vdash w_2 : \beta}{[\,;\,]\vdash \lambda\!\!\!^{\circ} x.w_2 : \alpha \multimap \beta} \qquad [\,;\,]\vdash v : \alpha}{[\,;\,]\vdash (\lambda\!\!\!^{\circ} x.w_2)v : \beta}}{[\,;\,]\vdash (w_1)(\lambda\!\!\!^{\circ} x.w_2)v : \gamma}$$

Nous voyons ici une différence, c'est que le type de w_1 peut être différent car dans un cas on a $\rightsquigarrow_1$ et dans l'autre cas on a nécessairement $\multimap$, puisque lors de l'élimination de cette flèche, il peut y avoir moins de variables linéaires libres (x est libre ou non selon le terme) dans l'argument. Donc nous ne pouvons pas conclure à une égalité des types principaux.

Cependant

- si le premier arbre est un arbre de typage principal pour $(\lambda\!\!\!^{\circ} x.(w_1)w_2)v$, on voit que $(w_1)(\lambda\!\!\!^{\circ} x.w_2)v$ est typable du même type car l'arbre de typage de celui-ci reste valide si l'on transforme $\rightsquigarrow_1$ en $\multimap$. Par conséquent il existe S_1 telle que $S_1\mathrm{TP}((w_1)(\lambda\!\!\!^{\circ} x.w_2)v)=\mathrm{TP}((\lambda\!\!\!^{\circ} x.(w_1)w_2)v)$, avec S_1 qui n'est pas nécessairement SL.
- Si le second arbre est un arbre de typage principal pour $(w_1)(\lambda\!\!\!^{\circ} x.w_2)v$, voyons les cas selon $\rightsquigarrow_1$:
  - $\rightsquigarrow_1=\multimap$: le terme $(\lambda\!\!\!^{\circ} x.(w_1)w_2)v$ est alors typable du même type et il existe S_2 telle que $\mathrm{TP}((w_1)(\lambda\!\!\!^{\circ} x.w_2)v)=S_2\mathrm{TP}((\lambda\!\!\!^{\circ} x.(w_1)w_2)v)$ et les deux types principaux sont égaux et on peut prendre l'identité pour S.
  - $\rightsquigarrow_1=\to$: c'est impossible. En effet nous savons que le terme est typable avec une flèche $\multimap$ pour w_1, donc il existe une substitution qui transforme la flèche du type de w_1 en $\multimap$ par définition du type principal. Or on ne peut pas transformer une flèche intuitionniste en flèche linéaire.
  - $\rightsquigarrow_1= -?_1$ une flèche sous-spécifiée. Notons alors S la substitution qui transforme $-?_1$ en $\multimap$.
    Si $-?_1$ n'appartient pas au type principal, comme on peut appliquer S à l'arbre et comme S laisse le type invariant, il existe S_2 telle que
    $\mathrm{TP}((w_1)(\lambda\!\!\!^{\circ} x.w_2)v)=S_2\mathrm{TP}((\lambda\!\!\!^{\circ} x.(w_1)w_2)v)$.
    Les deux termes ont alors même type principal et nous pouvons prendre l'identité pour S.
    Sinon, il existe S_2 telle que $S\mathrm{TP}((w_1)(\lambda\!\!\!^{\circ} x.w_2)v)=S_2\mathrm{TP}((\lambda\!\!\!^{\circ} x.(w_1)w_2)v)$.
    De plus nécessairement la substitution S_1 vue plus haut contient S dans ce cas, car l'arbre de typage de t se fait avec $\multimap$. Alors si l'on note S_1' la substitution identique à S_1 mais laissant $-?_1$ invariante,
    $S_1'\ S\mathrm{TP}((w_1)(\lambda\!\!\!^{\circ} x.w_2)v)=\mathrm{TP}((\lambda\!\!\!^{\circ} x.(w_1)w_2)v)$.
    Donc $S\mathrm{TP}((w_1)(\lambda\!\!\!^{\circ} x.w_2)v)=\mathrm{TP}((\lambda\!\!\!^{\circ} x.(w_1)w_2)v)$.

Finalement dans tous les cas : $S\mathrm{TP}((w_1)(\lambda\!\!\!^{\circ} x.w_2)v)=\mathrm{TP}((\lambda\!\!\!^{\circ} x.(w_1)w_2)v)$ avec S une substitution SL. □

Un dernier lemme :

Lemme 3.3.6 *Soit t un terme linéaire. Soit t' tel que $t \to_\beta t'$. Alors il existe une substitution S SL telle que TP(t)=S TP(t').*

Preuve : par induction sur la complexité de t. Comme t se réduit une fois, t n'est pas une variable.

– $t = \lambda x.u$: alors $t' = \lambda x.u'$ avec $u \to_\beta u'$ et u, u' vérifient la propriété, c'est à dire qu'il existe S SL telle que TP(u)=S TP(u'). Comme TP(t) (resp. TP(t')) est le même que TP(u) (resp. TP(u')) au déplacement du type de x près, on obtient alors TP(t)=S TP(t').

– $t = (u)\ v$: on a trois cas pour t' :

1. $u \to_\beta u'$ et $t' = (u')v$. Par induction il existe S SL telle que TP(u)=S TP(u'). Alors d'après le lemme 3.3.4, il existe S' telle que TP($(u)v$)=S' TP($(u')v$), d'où le résultat.
2. $v \to_\beta v'$ et $t' = (u)v'$. Par induction il existe S SL telle que TP(v)=S TP(v'). Alors d'après le lemme 3.3.4, il existe S' telle que TP($(u)v'$)=S' TP($(u)v'$), d'où le résultat.
3. $u = \lambda x.w$ et $t' = w[x := v]$. Regardons les cas sur w, qui ne peut pas être une constante car le terme est linéaire et contient nécessairement x une fois :
   (a) $w = x$: il est clair que TP(t)=TP(v).
   (b) $w = \lambda y.w_1$: nous utilisons le fait que si $(\lambda x \lambda y.w_1)v$ est typable alors il en est de même pour $\lambda y(\lambda x.w_1)v$, qui a le même type (peut se voir en faisant un arbre de typage pour chacun). Alors par induction il existe S SL telle que TP($(\lambda x.w_1)v$)=S TP($w_1[x := v]$). Alors pour les mêmes raisons que le cas vu plus haut, TP($\lambda y.(\lambda x.w_1)v$)=$S$ TP($\lambda y.w_1[x := v]$). Or le terme $\lambda y.w_1[x := v]$ est exactement $t' = w[x := v]$. D'où le résultat.
   (c) $w = \lambda\!\!\!^{\circ} y.w_1$: même chose que précédemment.
   (d) $w = (w_1)w_2$ avec x dans w_1 : par le lemme 3.3.5, nous savons que TP(t)=TP($((\lambda x.w_1)v)w_2$). Or par induction on sait qu'il existe S SL telle que
   $$\text{TP}((\lambda x.w_1)v) = \text{S TP}(w_1[x := v])$$
   Alors d'après le lemme 3.3.4, il existe S' SL telle que
   $$\text{TP}(((\lambda x.w_1)v)w_2) = S' \text{ TP}((w_1[x := v])w_2)$$
   Donc TP(t)=S' TP(t').
   (e) $w = (w_1)w_2$ avec x dans w_2 : c'est le même raisonnement que le cas précédent, en utilisant encore le lemme 3.3.5 et le lemme 3.3.4.
4. $u = \lambda\!\!\!^{\circ} x.w$ et $t' = w[x := v]$. C'est ici semblable au cas précédent, sauf pour le cas où x est dans w_2 à cause du lemme 3.3.5. En effet, celui-ci nous donne S S telle que TP(t)=S TP($(w_1)\ (\lambda\!\!\!^{\circ} x.w_2)v$). Or par induction on sait qu'il existe S_1 SL telle que
   $$\text{TP}((\lambda\!\!\!^{\circ} x.w_2)v) = S_1 \text{ TP}(w_2[x := v])$$
   Alors d'après le lemme 3.3.4, il existe S'_1 SL telle que
   $$\text{TP}((w_1)\ (\lambda\!\!\!^{\circ} x.w_2)v = \text{S}'_1 \text{ TP}((w_1)w_2[x := v])$$
   Donc TP(t)=S S'_1 TP($(w_1)w_2[x := v]$). Et S S'_1 est bien une substitution SL. □

Remarque 3.3.7

1. Nous avons réellement utilisé dans le lemme 3.3.6 le fait d'avoir une variable linéaire, car TP($(\lambda x.(w_1)w_2)v$) n'est pas égal à TP($((\lambda x.w_1)v)(\lambda x.w_2)v$) dans le cas général, ce que nous devrions prouver pour avoir le résultat dans un cas non linéaire. On peut vérifier par exemple que pour $w_1 = \lambda z.(z)\ (x)\xi\,\xi'$, $w_2 = \lambda z(x)\ \zeta\ z$ et $v = \lambda w.w$ les deux types sont distincts. Par ailleurs, si $(\lambda x.(w_1)w_2)v$ est typable alors $((\lambda x.w_1)v)(\lambda x.w_2)v$ est typable, mais la réciproque est fausse. Voir par exemple avec $w_1 = \lambda z.(z)\ (x)\xi$, $w_2 = \lambda z.(x)\ \zeta\ z$ et $v = \lambda w.w$.

2. On peut aussi se poser la question de savoir si le lemme 3.3.6 reste vrai avec des termes affines (dont les variables apparaissent au plus une fois). Mais la réponse est négative, car si l'on prend $t = \lambda z.(\lambda x.y)\ (z)u$, avec u un terme quelconque normal, on voit après β-réduction la disparition de la variable z, qui change alors de type.

Finissons enfin par la proposition promise :

Proposition 3.3.8 *Soit t un terme linéaire typable. Alors son type principal vérifie les points suivants :*
- *Chaque variable de type qui apparaît n'apparaît que deux fois, avec une occurrence positive et une négative ;*
- *La propriété SNIP est vérifiée ;*
- *Les flèches sous-spécifiées qui apparaissent sont uniques (elles sont toutes distinctes).*

Preuve : considérons $t^\star$ sa forme normale. On sait par le lemme 3.3.2 que $t^\star$ vérifie la propriété. Par induction sur la longueur de la réduction on prouve la proposition.
- $t = t^\star$: c'est clair par le lemme 3.3.2.
- $t \to_\beta t' \to_\beta^* t^\star$: t' vérifie la propriété par induction et le lemme 3.3.6 nous donne que $\mathrm{TP}(t)=_S \mathrm{TP}(t')$ avec S une substitution SL. Donc les variables de type sont identiques, les positions des flèches sont respectées et il y a moins de flèches sous-spécifiées, donc elles sont toujours distinctes. □

Remarque 3.3.9 Ces résultats ne sont vrais que pour le type principal des termes linéaires. Le fait que tous les atomes apparaissent deux fois en position opposée fait que si l'on transforme une variable de type en type flèche contenant une flèche $\to$, cette flèche sera alors à la fois en position positive et négative dans le type.

3.4 Termes η-longs

3.4.1 Définitions

Définissons ce qu'est un terme η-long et tout d'abord ce qu'est un arbre de typage η-long.

Définition 3.4.1 *Soit $\mathcal{T}$ un arbre de typage pour un terme t. $\mathcal{T}$ est η-long si la racine de $\mathcal{T}$ est dans l'un des cas suivants :*
- $[\Gamma\ ;\ \Delta] \vdash (x)\ t_1\ \ldots\ t_n : \alpha$ *avec $n \geq 0$ où α est une variable de type ou une constante et l'arbre de typage de chaque t_i est η-long ;*
- $[\Gamma\ ;\ \Delta] \vdash (c)\ t_1\ \ldots\ t_n : a$ *avec $n \geq 0$ et l'arbre de typage de chaque t_i est η-long ;*
- $[\Gamma\ ;\ \Delta] \vdash \lambda x.t' : \alpha \to \beta$ *et l'arbre de typage de t' est η-long ;*
- $[\Gamma\ ;\ \Delta] \vdash \text{⅋}x.t' : \alpha \multimap \beta$ *et l'arbre de typage de t' est η-long ;*
- $[\Gamma\ ;\ \Delta] \vdash (\text{⅋}x_1 \ldots x_m.t')\ t_1\ \ldots\ t_n : \alpha$ *avec α une variable de type ou une constante et t' n'est pas de la forme $\text{⅋}x.u$ et les arbres de typage de $\lambda x_1 \ldots x_m.t'$ et des t_k sont η-longs.*

Définition 3.4.2 *On dit qu'un terme t est η-long s'il a un arbre de typage η-long (donc en particulier il est typé).*

Nous avons les propositions suivantes

Lemme 3.4.3 *Soit t un terme η-long. Si $t = (\text{⅋}x_1 \ldots x_m.t')\ t_1\ \ldots\ t_n$ et t' n'est pas de la forme $\text{⅋}x.u$ alors $m = n$.*

Preuve : Comme $\text{⅋}x_1 \ldots x_m.t'$ est η-long, t' est η-long et comme il n'est pas de la forme $\text{⅋}x.u$, il est de type atomique α. Donc le type de $\text{⅋}x_1 \ldots x_m.t'$ s'écrit $A_1 \rightsquigarrow \ldots -?A_m \rightsquigarrow \alpha$. Comme t est η-long il est de type atomique et donc $m = n$. □

Remarque 3.4.4 En particulier, l'écriture d'un terme t η-long peut être simplifiée :
- $t = (x)\ t_1\ \dots\ t_n$;
- $t = (c)\ t_1\ \dots\ t_n$;
- $t = \lambda x.u$;
- $t = \lambda\!\!\!^{*}\, x.u$;
- $t = (\lambda\!\!\!^{*}\, x_1 \dots x_n.t')\ t_1\ \dots\ t_n$ avec t' n'étant pas de la forme $\lambda\!\!\!^{*}\, x.u$. Nous noterons parfois $t = (\lambda\!\!\!^{*}\, x_1 \dots x_n.(\dots)\dots)\ t_1\ \dots\ t_n$ un tel terme.

Le lemme suivant prouve que notre notion est identique à la définition habituelle de terme η-long.

Lemme 3.4.5 *Soit t un terme η-long. Soit t' un sous-terme de t ayant un type flèche. Alors t' est appliqué ou est de la forme $\lambda\!\!\!^{*}\, x.u$.*

Preuve : Par induction sur t
- Si $t = (x)\ t_1\ \dots\ t_n$: Si t' est un sous-terme de l'un des t_k, l'induction termine. Sinon t' est un $(x)\ t_1\ \dots\ t_k$ avec $k < n$. Donc t est appliqué.
- Si $t = (c)\ t_1\ \dots\ t_n$: idem.
- Si $t = \lambda x.u$: si t' est un sous-terme de u l'induction termine. Sinon $t' = t$ et est donc de la forme $\lambda x.u$.
- Si $t = \lambda\!\!\!^{*}\, x.u$: analogue.
- Si $t = (\lambda\!\!\!^{*}\, x_1 \dots x_n.t')\ t_1\ \dots\ t_n$: si t' est un sous-terme de $\lambda\!\!\!^{*}\, x_1 \dots x_n.t'$ ou de l'un des t_k l'induction termine. Sinon $t' = (\lambda\!\!\!^{*}\, x_1 \dots x_n.t')\ t_1\ \dots\ t_k$ avec $k < n$ et donc t' est appliqué. □

Lemme 3.4.6 *Soit $\mathcal{T}$ un arbre de typage d'un terme t, soit S une substitution. Si $S\mathcal{T}$ est η-long alors $\mathcal{T}$ est η-long.*

Preuve : Par induction sur t, en remarquant que si $S\tau$ est un atome alors τ est un atome. □

3.4.2 Algorithme 3

Ici, l'algorithme de typage est modifié afin d'être plus pratique à utiliser pour ce problème particulier. L'entrée est la même que l'algorithme 2, mais la sortie est un arbre de typage η-long. C'est à dire que l'algorithme génère un arbre de typage de manière identique à l'algorithme 2, puis vérifie que le résultat éventuel est η-long. Ainsi l'appel récursif de l'algorithme donne des arbres de typages η-longs.
Nous savons que dans certains cas il est impossible de trouver un arbre de typage η-long, donc dans les preuves par induction sur t η-long nous ne regarderons que les cas pour t décrits dans la remarque 3.4.4. Le seul cas qui change est celui de l'application $t = (\lambda\!\!\!^{*}\, x_1 \dots x_n.t')\ t_1\ \dots\ t_n$ dans laquelle le terme abstraction a un nombre de $\lambda\!\!\!^{*}$ de tête égal au nombre d'arguments.

Proposition 3.4.7 *Soit t un terme η-long. Alors l'algorithme 3 termine avec succès, c'est à dire donne un arbre η-long, identique à celui donné par l'algorithme 2.*

Preuve : Puisque t est typable et η-long, il est clair que l'algorithme termine avec succès (car les cas possibles sont donnés par la remarque 3.4.4) et donne un arbre de typage $\mathcal{T}_0$ qui est principal parce qu'il construit l'arbre de la même manière que l'algorithme 2. Comme t est η-long il existe un arbre $\mathcal{T}$ η-long. Comme $\mathcal{T}_0$ est principal, il existe S telle que $\mathcal{T} = S\mathcal{T}_0$ et donc par le lemme 3.4.6 $\mathcal{T}_0$ est η-long. □

Pour pouvoir atteindre notre objectif, nous allons introduire une fonction dont le but est intuitivement de justifier chaque élément, flèche ou atome, du type principal d'un terme. Essentiellement, cette fonction
- prend comme arguments
  - une adresse dans un type ;

– un terme.
- renvoie un ensemble de sous-termes qui ont un type ayant en tête l'élément pointé par l'adresse.

Il est en important de noter que si un type est principal, c'est que chacun des éléments qui le composent a une raison d'être. Dans le cas précis des termes η-longs il est possible d'attacher des sous-termes à chaque atome et chaque flèche du type principal d'un terme.
Ce qui permet celà est le fait que chaque sous-terme qui a un type flèche est soit une abstraction, soit appliqué à des arguments. Ainsi dans le second cas il est toujours possible de prendre les arguments.

Note 3.4.8 Dans le cas de termes quelconques, une telle tâche serait rendue difficile, car l'absence d'arguments à certains sous-termes ayant un type flèche sans être des abstractions implique que ceux-ci ont le même type que d'autres sous-termes qui peuvent être difficilement détectés dans le terme.

La fonction que nous allons définir est principalement utilisée avec des termes η-longs à des adresses de leur type principal, même si la définition est plus générale.

Nous devons d'abord introduire une notion d'adresse.

3.4.3 Adresses

Définition 3.4.9 *Une adresse dans un type est une liste finie, éventuellement vide, d'éléments de* $\{0, 1\}$. *Nous noterons* $[]$ *l'adresse vide,* $\epsilon :: c$ *la liste* c *à laquelle on a ajouté* ϵ *et* $c :: d$ *la concaténation de deux adresses. Pour plus de commodité, nous noterons* 1^k *une liste comportant* k 1.

Définition 3.4.10 *Soit* f *la fonction qui à une adresse* c *et un type* T *associe un sous-type de* T *définie ainsi :*
– $f([], T) = T$;
– $f(0 :: c, T_1 \rightsquigarrow T_2) = f(c, T_1)$;
– $f(1 :: c, T_1 \rightsquigarrow T_2) = f(c, T_2)$;
– *Dans les autres cas,* f *n'est pas définie.*

Définition 3.4.11 *On dit que l'adresse* c *est une adresse dans le type* T *si* $f(c, T)$ *est défini. Dans ce cas on dit que* $f(c, T)$ *est le sous-type de* T *à l'adresse* c.

Définition 3.4.12 *Soit* c *une adresse dans le type* T. *One note* $h(c, T)$ *la tête du type* $f(c, T)$ *(qui est soit une flèche soit un atome).*

Définition 3.4.13 *Si* c *est une adresse dans le type* T *on dit que* c *est positif si* c *a un nombre pair de* 0 *et que* c *est négatif dans le cas contraire.*

Lemme 3.4.14 *Soit* T *un type et soit* T' *un sous-type de* T. *Alors il existe une adresse* c *telle que* $f(c, T) = T'$.

Preuve : par induction sur la complexité de T. Si $T' = T$, il suffit de prendre $S = []$. Sinon $T = T_1 \rightsquigarrow T_2$ et T' est un sous-type de T_1 (ou de T_2). Par hypothèse d'induction, il existe c tel que $f(c, T_1) = T'$ (resp. $f(c, T_2) = T'$).
Alors $T' = f(0 :: c, T)$ (resp. $T' = f(1 :: c, T)$). □

Lemme 3.4.15 *Soit* c *l'adresse d'un type* T. *Alors* c *a la même polarité que* $f(c, T)$ *dans* T.

Preuve : par induction sur la taille de c.
– $c = []$: clair.

– $c = 0 :: c'$: on a $T = T_1 \rightsquigarrow T_2$ et $f(c,T)$=$f(c',T_1)$. Par induction c' a la même polarité que $f(c',T_1)$ dans T_1. Or c' a la polarité opposée à c et T_1 a la polarité opposée à T dans T. Donc c a bien la polarité de $f(c,T)$ dans T.
– $c = 0 :: c'$: on a $T = T_1 \rightsquigarrow T_2$ et $f(c,T)$=$f(c',T_2)$. Par induction c' a la même polarité que $f(c',T_2)$ dans T_2. Or c' a la même polarité que c et T_2 a la même polarité que T dans T. Donc c a bien la polarité de $f(c,T)$ dans T. □

Lemme 3.4.16 *Soit c une adresse dans un type T et S une substitution. Alors c est une adresse dans ST.*

Preuve : Par induction sur la taille de c
– Si $c = []$, alors c est toujours une adresse dans un type.
– Si $c = 0 :: c'$, alors $T = T_1 \rightsquigarrow T_2$ et c est une adresse dans T_1. Par induction c est une adresse dans ST_1, mais $ST = ST_1\ S \rightsquigarrow\ ST_2$, donc c est bien une adresse dans ST.
– Si $c = 1 :: c'$ alors le raisonnement est similaire au cas précédent. □

Lemme 3.4.17 *Soit c une adresse dans un type T et dans un type T'. Supposons que $h(c,T)$ soit identique à $h(c,T')$. Soit S une substitution. Alors $h(c,ST)$ est identique à $h(c,ST')$.*

Preuve : Par induction sur la taille de c.
– Si $c = []$, alors la tête de T est la même que celle de T', notons la h. Alors leurs images par S ont même tête : Sh.
– Si $c = 1 :: c'$, alors $T = T_1 \rightsquigarrow T_2$ et $T' = T'_1 \rightsquigarrow T'_2$. Par hypothèse et définition de f et h, $h(c',T_2)$ est identique à $h(c',T'_2)$. Alors par induction $h(c',ST_2)$ est identique à $h(c',ST'_2)$. Comme $ST = ST_1\ S \rightsquigarrow\ ST_2$ et $ST' = ST'_1\ S \rightsquigarrow\ ST'_2$ nous avons bien le résultat.
– Si $c = 0 :: c'$, le raisonnement est analogue. □

3.4.4 Termes justifiants

Définition 3.4.18 *Nous définissons $\mathcal{X}x.E$ où E est un ensemble de termes et $\mathcal{X}$ est fixé l'ensemble des termes $\mathcal{X}x.t$ pour t dans E.*

Note 3.4.19 Eventuellement dans un tel ensemble $\mathcal{X}x.E$ le nom de la variable liée par l'abstraction peut changer d'un terme à l'autre. Par exemple $\{\lambda y.t_1, \lambda z.t_2\}$ est également considéré comme un $\mathcal{X}x.E$. Le nom des variables liées étant peu important, nous considérons pour simplifier en dehors des exemples qu'il s'agit de la même variable qui est liée dans les termes de E.

Définition 3.4.20 *Définissons φ la fonction définie sur les couples (adresse,ensemble de termes) par induction sur la taille des adresses :*
– $\varphi([],E)$=$(E,[],E)$;
– $\varphi(1 :: c, \mathcal{X}x.E)$=$\varphi(c,E)$;
– $\varphi([0], \mathcal{X}x.E)$=$(\{x\},[0],\mathcal{X}x.E)$;
– $\varphi(0 :: 1^k :: 0 :: c, \mathcal{X}x.E)$=$\varphi(c,F)$ $(k \geq 0)$ *(⋆)* ;
– $\varphi(0 :: 1^k, \mathcal{X}x.E)$=$(G, 0 :: 1^k, \mathcal{X}x.E)$ $(k \geq 1)$ *(⋆⋆)* ;
– *φ n'est pas définie dans les autres cas.*
(⋆) où $F = \{t_{k+1} \mid (x)t_1 \ldots t_{k+1}$ est un sous-terme d'un terme de $E\}$. Si F est vide alors φ n'est pas définie.
(⋆⋆) où $G = \{(x)\ t_1\ \ldots\ t_k \mid (x)\ t_1\ \ldots\ t_k$ est un sous-terme d'un terme de $E\}$. Si G est vide alors φ n'est pas définie.

Remarque 3.4.21 Dans la définition 3.4.20 nous avons séparé les cas $c = [0]$ avec le cas $c = 0 :: 1^k$ pour $k \geq 1$. La raison est que dans le deuxième cas, on cherche les occurrences de x mais pas dans le premier cas. En effet si $c = [0]$, ce qui nous intéresse c'est de justifier ce qui est à gauche de la flèche du type de termes de la forme $\mathcal{X}x.t$. Il est clair que c'est bien x qui

le justifie même si x n'a pas d'occurrence. Dans l'autre cas, c'est le type d'un des arguments de x qui nous intéresse, donc on a besoin des occurrences de x.

Nous avons besoin de définir des notations pour simplifier les explications.

Définition 3.4.22 *La fonction φ renvoie un triplet.*
- *Le premier élément est appelé ensemble des sous-termes justifiants.*
- *Le second élément est appelée adresse du dernier appel.*
- *Nous appelons termes du dernier appel le dernier élément.*
- *Le couple formé par l'adresse du dernier appel et les termes du dernier appel est appelé le dernier appel à φ.*

Remarque 3.4.23 Le terme "dernier appel" provient du fait que φ est une fonction récursive, donc le dernier appel correspond au dernier appel récursif. Il est à noter cependant que si la récursivité se fait sur la forme de l'adresse, ce n'est pas une définition portant sur les trois cas $[]$, $0 :: c$ et $1 :: c$ avec un appel récursif sur c. Ceci implique le fait que parfois φ n'est pas définie. Il est toutefois évident qu'à chaque appel récursif la taille de l'adresse a diminué.

Définition 3.4.24 *Il y a trois cas dans la définition de φ qui donnent un résultat et pour chacun l'adresse du dernier appel est distincte.*
- *Si cette adresse est $[]$, nous dirons que φ renvoie des termes η-longs.*
- *Si l'adresse est $[0]$, nous dirons qu'elle renvoie des variables.*
- *Si l'adresse est de la forme $0 :: 1^k$ avec $k \geq 1$ nous dirons qu'elle renvoie des termes de la forme $(x)\ t_1\ \ldots\ t_k$.*

Définition 3.4.25 *Soit la fonction $\tilde{\varphi}$ définie sur les triplets (adresse,variable,ensemble de termes) par $\tilde{\varphi}(c, x, E) = \varphi(0 :: c, \not{\lambda} x.E)$ où $\not{\lambda}$ est $\not{\lambda}$ si x est une variable linéaire dans les termes de E ou par λ si elle est intuitionniste.*

Voici un exemple permettant de concrétiser la définition de φ :

Exemple 3.4.26 Donnons

$$t = \lambda x.(f) \quad (x)t_1\ \lambda y.y \quad (x)t_2\ \lambda z.t_3$$

avec t_1, t_2 et t_3 des termes quelconques. Prenons $c = [0, 1, 0, 0]$. Alors

$$\varphi(c, \{t\}) = \varphi([0], \{\lambda y.y, \lambda z.t_3\}) = (\{y, z\}, [0], \{\lambda y.y, \lambda z.t_3\})$$

Nous devons donner une explication pour la dernière égalité. Dans la définition de φ nous écrivions $\varphi(c, \not{\lambda} x.E)$, l'abstraction liant la variable x dans l'ensemble des termes de E. mais nous avons ici un ensemble $\{\lambda y.y, \lambda z.t_3\}$, le nom des variables liées étant différent. Le nom des variables liées n'a pas de réelle d'importance, mais pour une raison de lecture, il peut être préférable de les différencier. Ainsi nous donnons l'ensemble $\{y, z\}$ comme ensemble de sous-termes justifiants.
remarquons que le type de t est de la forme $T = (\alpha \to (\beta \to \beta) \to \gamma) \to \delta$. On a $f(c, T) = \beta$ et les deux variables y et z ont pour type β. Donc l'ensemble $\{y, z\}$ est bien un ensemble de termes justifiant l'atome β.

Nous allons avoir besoin des lemmes suivants. Des lemmes similaires pour $\tilde{\varphi}$ peuvent être prouvés de même manière. Certains permettent de justifier le vocabulaire que nous avons donné dans les définitions 3.4.22 et 3.4.24. La plupart du temps nous les utiliserons de manière implicite.

Lemme 3.4.27 *Soit E un ensemble de termes. Soit c une adresse. Si $\varphi(c, E)$ est défini, notons (c^*, E^*) son dernier appel. Alors $\varphi(c, E) = \varphi(c^*, E^*)$.*

Preuve : par induction sur la taille de l'adresse. Le seul cas non trivial est le cas où $c = 1 :: c'$. Dans ce cas $E = \lambda\!\!\!^{*}x.E'$ et $\varphi(c, E) = \varphi(c', E')$. Le dernier appel de $\varphi(c', E')$ étant le même, l'induction termine. □

Lemme 3.4.28 *Soit E un ensemble de termes η-longs. Soit c une adresse. Si $\varphi(c, E)$ est défini alors les termes du dernier appel sont η-long. En particulier, si l'adresse du dernier appel est $[]$ alors les sous-termes justifiants sont η-longs.*

Preuve : par induction sur la taille de l'adresse. Le seul cas non trivial est le cas où $c = 1 :: c'$. Dans ce cas $E = \lambda\!\!\!^{*}x.E'$ et $\varphi(c, E) = \varphi(c', E')$. Puisque les termes de E sont η-longs, il en est de même pour ceux de E'. L'induction permet donc de terminer. □

Lemme 3.4.29 *Soit E et F deux ensembles de termes. Soit c une adresse. Si $\varphi(c, E)$ et $\varphi(c, F)$ sont définis alors l'adresse du dernier appel pour E est la même que l'adresse du dernier appel pour F.*

Preuve : par induction sur la taille de l'adresse. Le seul cas non trivial est le cas où $c = 1 :: c'$. Dans ce cas $E = \lambda\!\!\!^{*}x.E'$ et $\varphi(c, E) = \varphi(c', E')$. De même on a $F = \lambda\!\!\!^{*}x.F'$ et $\varphi(c, F) = \varphi(c', F')$. L'induction termine. □

Lemme 3.4.30 *Soit E et F deux ensembles de termes. Soit c une adresse. Si $\varphi(c, E)$ et $\varphi(c, F)$ sont définis alors $\varphi(c, E \cup F)$ est défini. De plus, l'ensemble des sous-termes justifiant de $\varphi(c, E \cup F)$ est l'union des deux ensembles justifiants de $\varphi(c, E)$ et $\varphi(c, F)$, et il en est de même pour les termes du dernier appel.*

Preuve : par induction sur la taille de l'adresse. Le seul cas non trivial est le cas où $c = 1 :: c'$. Dans ce cas $E = \lambda\!\!\!^{*}x.E'$ et $\varphi(c, E) = \varphi(c', E')$. De même on a $F = \lambda\!\!\!^{*}x.F'$ et $\varphi(c, F) = \varphi(c', F')$. Alors $\varphi(c, E \cup F) = \varphi(c', E' \cup F')$ et l'induction termine. □

Remarque 3.4.31 Le lemme 3.4.30 ci-dessus permet de justifier que si l'on a un terme t de la forme $(x)\, t_1 \ldots t_n$, et si c est une adresse dans le type d'une variable y, alors l'ensemble des sous-termes justifiants de $\tilde{\varphi}(c, y, \{t\})$ contient l'ensemble des sous-termes justifiants de chacun des $\tilde{\varphi}(c, y, \{t_i\})$. Il suffit pour voir cela de regarder la définition de $\tilde{\varphi}$, puis de φ, que l'on se trouve bien dans le cas du lemme 3.4.30.

Les deux lemmes ci-dessous permettent d'expliquer l'appellation 'sous-termes justifiants'.

Lemme 3.4.32 *Soit E un ensemble de termes typés. Soit c une adresse telle que $\varphi(c, E)$ est défini. Soit un atome (resp. une flèche) tel que, pour tout type T de terme de E, $h(c, T)$ est cet atome (resp. flèche). Alors les sous-termes justifiants ont un type ayant cet atome (resp. flèche) en tête.*

Preuve : par induction sur la taille de l'adresse. Traitons le cas où $c = 1 :: c'$. Dans ce cas $E = \lambda\!\!\!^{*}x.E'$ et $\varphi(c, E) = \varphi(c', E')$. Tout terme t de E s'écrit $\lambda\!\!\!^{*}x.t'$ et si t est de type T et t' de type T' alors $h(c, T) = h(c', T')$ et l'induction termine. □

Lemme 3.4.33 *Soit E un ensemble de termes ayant le même type T. Soit c une adresse telle que $\varphi(c, E)$ est défini. Alors les sous-termes justifiants ont tous le même type, à savoir $f(c, T)$.*

Preuve : par induction sur la taille de l'adresse. Traitons le cas où $c = 1 :: c'$. Dans ce cas $E = \lambda\!\!\!^{*}x.E'$ et $\varphi(c, E) = \varphi(c', E')$. Tout terme t de E s'écrit $\lambda\!\!\!^{*}x.t'$ et si t est de type T et t' de type T' alors $f(c, T) = f(c', T')$ et l'induction termine. □

Lemme 3.4.34 *Soit E un ensemble de termes. Soit c et d des adresses. Si $\varphi(c, E)$ est défini, alors en notant (c', E') son dernier appel, nous avons l'égalité suivante :*

$$\varphi(c :: d, E) = \varphi(c' :: d, E')$$

Preuve : par induction sur la taille de l'adresse, de manière évidente. □

3.4.5 Classes

Nous avons besoin d'outils permettant de suivre certaines propriétés durant le typage d'un terme t. La difficulté provient du typage de l'application, pendant lequel il y a par deux fois une unification de types à faire. Pendant ces étapes nous possédons plusieurs jugements de termes, et il faut unifier le type des variables communes à chacun d'eux. Pour cela il faut chercher les parties des types qui ne coïncident pas et les unifier.
Pour pointer ces parties de type, nous allons définir les classes qui, évidemment, vont utiliser les adresses définies précédemment. Elles regroupent intuitivement tous les termes qui concordent à la même adresse.
A la fin de l'unification, les classes regroupent alors tous les termes, puisque les types des variables sont tous identiques.
Comme la preuve de la proposition 3.4.58, portant sur une propriété des types des termes η-longs, se fera par induction, nous devrons prouver qu'une propriété semblable reste stable pendant l'unification des types des variables.

Note 3.4.35

1. Dans un ensemble de jugements nous noterons toujours t_i $(1 \leq i \leq n)$ les termes. Un indice i est appelé un numéro de jugement. Nous noterons $\tau_i(x)$ (resp. $\tau(t)$) le type de la variable x dans le terme t_i (resp. du terme t).
2. Nous ne supposerons pas par la suite qu'un ensemble de jugements est un ensemble de jugements nécessairement valides. Il arrivera en effet que parfois nous définissions des jugements pour lesquels les variables ou le terme n'ont pas un bon type au regard des autres éléments du jugement.

Définition 3.4.36 (Classes) *Soit J un ensemble de jugements. Une classe C de J est :*
– soit (c, i) si c est une adresse dans $\tau(t_i)$;
– soit (c, x, I) si $I \subset \{1, \ldots, n\}$ est un ensemble de numéros de jugements qui est :
  – cohérent pour c et x : il existe un atome (ou une flèche) tel que, pour tout i dans I, x est libre dans t_i, c est une adresse dans $\tau_i(x)$ et $h(c, \tau_i(x))$ est cet atome (ou cette flèche),
  – maximal : I est un ensemble cohérent maximum pour la relation d'inclusion.

Définition 3.4.37 *Soit J un ensemble de jugements et C une classe de J.*
– si C est de la forme (c, i) on dit que c'est une classe de terme.
– si C est de la forme (c, x, I) on dit que c'est une classe de la variable x.
Dans les deux cas on dit que c est l'adresse de C.

Définition 3.4.38 (Point d'une classe) *Soit J un ensemble de jugements et C une classe de J. Soit $i \in \{1, \ldots, n\}$.*
– si $C = (c, x, I)$ on dit que $P = (c, x, i)$ est un point de C si $i \in I$.
– si $C = (c, i)$, nous disons également que $P = (c, i)$ est un point de C
Dans les deux cas on dit alors que C est la classe de P, et nous notons $C = Cl_J(P)$ ou simplement $Cl(P)$ s'il n'y a pas de confusion possible. On dit que P est un point de J.

Remarque 3.4.39 Dans la définition de point d'une classe, nous disons que C est la classe de P, et non une classe, car la maximalité nous impose l'unicité. Chaque élément (c, x, i) vérifiant que c est une adresse dans $\tau_i(x)$ est le point d'une classe par définition de la classe. De même pour (c, i) si c est une adresse dans $\tau(t_i)$ bien sûr.

Définition 3.4.40 (Ensemble associé à une classe) *Soit J un ensemble de jugements, et C une classe de J. L'ensemble associé à C, noté $E_J(C)$ (ou $E(C)$ si aucune confusion n'est possible) est*
– $\{t_i\}$ si $C = (c, i)$;
– $\{t_i \mid i \in I\}$ si $C = (c, x, I)$.

Définition 3.4.41 *Soient J et J' deux ensembles de jugement des mêmes termes t_i. On dit que C, classe de J, est incluse dans C', classe de J', si les deux ont même adresse, sont des classes de la même variable et si $E_J(C)$ est inclus dans $E_{J'}(C')$.*

Définition 3.4.42 (Tête de type d'une classe) *Soit J un ensemble de jugements, soit C une classe de J. On définit $H_J(C)$ (ou $H(C)$ lorsqu'il n'y a pas de confusion possible), tête de type de C, comme étant :*
– $h(c, \tau(t_i))$ si $C = (c, i)$;
– $h(c, \tau_i(x))$ si $C = (c, x, I)$ pour un $i \in I$.
Si P est un point de J, nous utiliserons la notation $H(P)$ pour $H(Cl(P))$.

Remarque 3.4.43

1. La définition de tête de type ne dépend pas du i choisi dans I, c'est à dire du P choisi, grâce à la cohérence.
2. Soit J un ensemble de jugement. Alors pour tout atome (resp. toute flèche) qui apparaît dans les types, il existe une classe dont la tête de type est cet atome (resp. flèche). En effet, on peut associer une adresse, un terme (et éventuellement une variable) qui pointe sur cet atome (resp. flèche) par le lemme 3.4.14, par conséquent la classe en découle. Ainsi tout atome ou flèche peut être vu comme un $H(C)$ pour une classe C.

Définition 3.4.44 (Singularité) *Soit J un ensemble de jugements. Soit a un type atomique (resp. $\rightsquigarrow$ une flèche) présent dans J. On dit que a (resp. $\rightsquigarrow$) est singulier dans J si il existe une unique classe C de J qui vérifie $H(C) = a$ (resp. $H(C) = \rightsquigarrow$).*

Définition 3.4.45 (Polarité) *Soit J un ensemble de jugement.*
Soit C une classe de J d'adresse c. Alors
– si $C = (c, i)$, $H(C)$ est positif si c est positive, négative sinon ;
– si $C = (c, x, I)$, $H(C)$ est positif si c est négatif, positif sinon.

Remarque 3.4.46 La définition de singularité est une généralisation de la définition d'unicité. Elle est équivalente dans le cas où l'ensemble de jugement ne comporte qu'un seul terme. La polarité est également équivalente à celle rencontrée plus haut dans le cas où il n'y a qu'un seul terme.

Définition 3.4.47 (Ensemble justifiant d'une classe) *Soit J un ensemble de jugements. Soit C une classe de J d'adresse c.*
– si C est une classe de terme, $\Phi(C) = \varphi(c, E(C))$.
– si C est une classe d'une variable x, $\Phi(C) = \tilde{\varphi}(c, x, E(C))$.
– une classe C est justifiée si $\Phi(C)$ est définie.

Exemple 3.4.48 Considérons $t_1 = \lambda y(x)\ y$ et $t_2 = \lambda w \parr z(x) \quad (w)\ z$. Considérons J l'ensemble de jugements comportant les jugements de types principaux de t_1 et t_2, à savoir :

$$x : a \multimap ?b \vdash t_1 : a \to b$$

et

$$x : a \multimap b \vdash t_2 : (c \multimap a) \to c \multimap b$$

Nous avons conservé volontairement des variables de type communes aux deux. La variable x peut être soit linéaire, soit intuitionniste. Considérons qu'elle est intuitionniste dans la suite. Alors :
$C_1 = ([], x, \{1\})$, $C_2 = ([0], x, \{1, 2\})$ et $C_3 = ([0, 1], 2)$ sont des classes de J.
Voyons cela de plus près :
– $h([], \tau_1(x)) = -?$ et $h([], \tau_2(x)) = \multimap$ est différent, donc C_1 est (cohérente et) maximale et $H(C_1) = -?$.
D'autre part, on peut vérifier que $\Phi(C_1) = (\{x\}, [0], \lambda x.t_1\})$.

– $h([0], \tau_1(x) = a$ et $h([0], \tau_2(x) = a$, donc C_2 est cohérente (et maximal) et $H(C_2) = a$.
D'autre part, on peut vérifier que $\Phi(C_2) = (\{(w)z, y\}, [], \{(w)z, y\})$.
– $H(C_3) = h([0, 1], \tau(t_2)) = a$.
D'autre part, on vérifie que $\Phi(C_3) = (\{(w)z\}, [0, 1], \{t_2\})$.

Lemme 3.4.49 *Soit J un ensemble de jugements, soit S une substitution et P un point de J. Alors P est un point de SJ et la classe de P dans J est incluse dans la classe de P dans SJ.*

Preuve : Le fait que P soit un point de SJ provient du lemme 3.4.16. Si $P = (c, x, i)$, alors $E(\mathrm{Cl}_{SJ}(P))$ est par définition l'ensemble des k qui vérifient que $h(c, S\tau_k(x))$=$h(c, S\tau_i(x))$. Or par définition de la classe de P dans J tous les termes t_k de la classe de P dans J vérifient $h(c, \tau_k(x))$=$h(c, \tau_i(x))$.
Donc en appliquant la même substitution S on a bien $h(c, S\tau_k(x))$=$h(c, S\tau_i(x))$ par le lemme 3.4.17, d'où le résultat. □

3.4.6 Des lemmes préliminaires

Nous avons besoin de lemmes pour alléger les preuves suivantes.

Définition 3.4.50 (Propriété des classes) *Soit t un terme η-long. Soit J un ensemble de jugements.*
Nous disons que J vérifie la propriété des classes si il vérifie les points suivants :
– *Les termes t_i de J sont dans l'un des cas suivants :*
  – *ils sont tous égaux à t,*
  – *$t = (x)\ t_1\ \ldots\ t_n$,*
  – *$t = (c)\ t_1\ \ldots\ t_n$,*
  – *$t = (\lambda\!\!\!^{*} x_1 \ldots x_n.t')\ t_2\ \ldots\ t_n$ avec $t_1 = \lambda\!\!\!^{*} x_1 \ldots x_n.t'$ et t' n'est pas une abstraction ;*
– *Toute classe C est justifiée ;*
– *Si le dernier appel de $\Phi(C)$ est de la forme $([0], \{\lambda\!\!\!^{*} x.u\})$ avec $x \notin u$ alors $H(C)$ est un atome a et a est singulier ;*
– *Si $H(C)$ est une flèche sous-spécifiée alors cette flèche est singulière et elle est négative ;*
– *Si $H(C) = \rightarrow$ alors cette flèche est positive.*

Remarque 3.4.51

1. Le premier point de la propriété des classes implique, du fait que t est η-long, que les t_i sont également η-longs. D'autre part il est important de remarquer que ce premier point ne dépend pas des types donnés dans J.
2. Cette définition est une généralisation de la propriété SNIP, car on peut voir en effet qu'elle est présente avec les deux derniers points.
3. La condition sur les classes qui sont telles que le dernier appel est de la forme $([0], \{\lambda\!\!\!^{*} x.u\})$ peut paraître étrange. Elle est cependant très importante.

   En effet, ces atomes, et ceux-là seulement, sont susceptibles d'être transformés, pendant les phases d'unification, en types complexes (on conserve le caractère η-long puisque les variables n'apparaissent pas par hypothèse). La condition d'unicité (ou de singularité ici) permet d'assurer que l'application de la substitution de cet atome par le type complexe conserve la propriété sur la polarité des flèches. En effet on est certain en particulier que cet atome n'apparaît pas dans une polarité différente, ce qui inverserait les polarités de toutes les flèches en plus de les multiplier.

   Dans le processus d'unification, en dehors de ces atomes particuliers, l'unification transforme toujours des atomes en atome, et des flèches en flèches, car les termes sont η-longs, et donc les types des variables qui apparaissent ont toujours globalement la même forme.

Lemme 3.4.52 *Soit J un ensemble de jugements vérifiant la propriété des classes. Supposons qu'il existe un point $P_1 = (c, x, i)$ tel que $f(c, \tau_i(x))$ est une variable de type α et un point $P_2 = (c, x, j)$ tel que $f(c, \tau_j(x))$ est un type non atomique T. Soit S la substitution qui transforme α en T. Alors SJ vérifie la propriété des classes.*

Preuve : Prenons $C_1 = \mathrm{Cl}_J(P_1)$ et $C_2 = \mathrm{Cl}_J(P_2)$, qui sont justifiées par hypothèse.

- Prouvons tout d'abord que $\mathrm{H}(C_1) = \alpha$ est singulier dans J.
  Dans le terme t η-long associé à J, les sous-termes justifiants pour C_1 et pour C_2 ont tous même type grâce au lemme 3.4.30, à la remarque 3.4.31 et au lemme 3.4.33.
  L'adresse du dernier appel pour les deux classes est la même par le lemme 3.4.29, par conséquent nous avons trois cas sur les sous-termes justifiants *a priori*
  - Ce sont des termes η-long. Cela signifie que pour C_2 ils sont de la forme $\lambda\!\!\!^{*}x.u$, mais pas pour C_1. De plus pour C_1 ces termes ne sont pas appliqués. Or dans t ils ont même type et restent η-longs. C'est impossible.
  - Ce sont des termes de la forme $(x)\ t_1\ \ldots\ t_k$. Cela signifie que dans C_2 ces termes sont appliqués, mais pas dans C_1. Or dans t ils ont même type et comme t est η-long c'est impossible.
  - Donc ce sont des variables.
    Alors dans C_2 certaines de ces variables sont appliquées, car comme elles ont un type flèche, il suffit d'allonger l'adresse par 1 par exemple et d'obtenir un point $P_2' = (c :: [1], x, j)$ dont la classe sera justifiée par hypothèse et donc renvoie des $(y)\ v_1$. En effet, puisque les termes obtenus sont des variables, le dernier appel est par définition de la forme $([0], \lambda\!\!\!^{*}y.E)$. Grâce au lemme 3.4.27, $\Phi(C_2) = \varphi([0], \lambda\!\!\!^{*}y.E)$. Puis grâce au lemme 3.4.34, $\Phi(\mathrm{Cl}_J(P_2')) = \varphi([0, 1], \lambda\!\!\!^{*}y.E)$. Enfin par définition, nous voyons que l'ensemble des sous-termes justifiant contient bien des $(y)\ v_1$.
    Mais dans C_1 aucune variable n'apparaît. En effet sinon elles ne seraient pas appliquées car elles ont pour type un atome et comme dans t ces variables ont toutes le même type, c'est nécessairement un type flèche à cause de C_2 (les variables étant appliquées), mais alors il y aurait des variables non appliquées de type flèche, ce qui est impossible.
    Nous obtenons donc par la propriété des classes le fait que α est singulier dans J. En effet, nous venons de montrer que nous avons une classe, C_1, telle que le dernier appel de $\Phi(C_1)$ est de la forme $([0], \{\lambda\!\!\!^{*}x.u\})$ avec $x \notin u$. Ainsi par hypothèse $\mathrm{H}(C_1) = \alpha$ est un atome, ce que nous savons, et est singulier.
- Prouvons maintenant que pour toute classe C_s de SJ il existe un point P_a de C_s tel que P_a est aussi un point de J et si $\mathrm{H}_{SJ}(C_s)$ est une flèche alors $\mathrm{H}_J(P_a)$ est la même flèche.
  Prenons P un point quelconque de C_s. Si c'est déjà un point de J alors c'est clair. C'est le cas pour toutes les classes (e, k), car α est singulier et donc le type des termes n'est pas modifié par S.
  Sinon notons (e, y, k) le point P. Soit $\dot{e}$ la sous-adresse la plus longue de e telle que pour toute sous-adresse stricte e' de $\dot{e}$ (e', x, j) est un point de J (au pire, $\dot{e} = []$). L'adresse $\dot{e}$ est différente de e. Considérons $h(\dot{e}, \tau(t_k))$, il y a trois cas *a priori*
  - c'est une flèche. Impossible car sinon nous pourrions prendre $\dot{e}$ plus grande car la flèche n'est pas transformée par S.
  - C'est une variable de type différente de α. Comme elle n'est pas transformée par S, nous aurions $e = \dot{e}$ car il est impossible d'allonger cette adresse dans SJ, ce qui est impossible.
  - Donc c'est la variable de type α. Nécessairement $\dot{e}$ est c et la classe du point $(\dot{e}, y, k)$ est C_1 par singularité de α et donc $y = x$. Alors $e = c :: c'$, c' étant une adresse dans le type T, car α est transformé en T. Le triplet (e, x, j) est un point de J car $P_2 = (c, x, j)$ est un point tel que $f(c, \tau_j)$ est T par hypothèse. Nous pouvons donc prendre $P_a = (e, x, j)$. Ce point est un point de SJ, qui est dans la classe de P, soit C_s par définition de la classe.
- Prouvons finalement que SJ vérifie la propriété des classes.
  - Le premier point reste toujours vrai, puisqu'il ne dépend pas des types donnés dans J comme nous l'avons déjà remarqué.
  - Toutes les classes sont justifiées grâce à ce que nous venons de prouver. En effet, soit C_s

une classe dans SJ. Alors il existe un point P_a dans C_s tel que P_a soit aussi un point de J. Or par hypothèse la classe de P_a dans J est justifiée, donc dans SJ aussi (la justification ne dépend pas du type).

- Si C_s est telle que $\Phi(C_s)$ est de la forme $([0], \{\not\lambda x.u\})$ avec $x \notin u$ alors il en est de même pour la classe C de P_a dans J. Alors par hypothèse $\mathrm{H}_J(C)$ est un atome b et b est singulier dans J. L'image de cet atome par la substitution est $\mathrm{H}_{SJ}(C_s)$.
  Si $\mathrm{H}_{SJ}(C_s)$ n'est pas un atome, c'est que $b = \alpha$ et donc P_2 est dans C_s par singularité de α. Or $\mathrm{H}(P_2)$ est une flèche par hypothèse donc on ne peut pas avoir $x \notin u$ pour tous les termes du dernier appel (il suffit de prendre comme plus haut la classe du point $(c :: 1, x, t_j)$ qui est justifiée pour voir que certaines des variables sont appliquées donc en particulier apparaissent).
  Donc on sait que $\mathrm{H}_{SJ}(C_s)$ est l'atome b (comme il est différent de α il n'est pas transformé par S). Supposons qu'il ne soit pas singulier dans SJ. Il existe donc une classe C'_s distincte de C_s telle que $\mathrm{H}_{SJ}(C_s)$=$\mathrm{H}_{SJ}(C'_s)$=b. Or il existe P'_a un point de C'_s qui est aussi un point de J. Puisque les classes de P_a et P'_a dans SJ sont distinctes (ce sont C_s et C'_s), elles le sont également dans J.
  Mais comme b est singulier dans J, $\mathrm{H}_J(\mathrm{Cl}(P_a)) = b$ est différent de $\mathrm{H}_J(\mathrm{Cl}(P'_a))$, qui doit être transformé en b par S par hypothèse sur C'_s. Or rien n'est transformé en b, donc c'est impossible.
  Donc b est singulier dans SJ.
- Si $\mathrm{H}(C_s)$ est une flèche sous-spécifiée alors il en est de même pour $\mathrm{H}(P_a)$ (P_a étant toujours le point dont l'existence a été prouvée plus haut) et cette flèche est singulière et négative dans J. C'est aussi une flèche négative dans SJ car il s'agit de la même adresse. Et elle est aussi singulière dans SJ par un argument analogue au paragraphe ci-dessus si l'on suppose qu'elle n'est pas singulière.
- Si $\mathrm{H}(C_s)$ est une flèche intuitionniste alors il en est de même pour $\mathrm{H}(P_a)$ et cette flèche est positive dans J, donc dans SJ aussi. □

Voici le même lemme mais dans le cas de la substitution d'une flèche.

Lemme 3.4.53 *Soit J un ensemble de jugements vérifiant la propriété des classes. Supposons qu'il existe un point $P_1 = (c, x, i)$ tel que $H(P_1)$ soit une flèche $-?_1$ sous-spécifiée et un point $P_2 = (c, x, j)$ tel que $H(P_2)$ est une autre flèche $\rightsquigarrow_2$ (soit sous-spécifiée, soit linéaire). Soit S la substitution qui transforme $-?_1$ en $\rightsquigarrow_2$. Alors SJ vérifie la propriété des classes.*

Preuve :

- Il est tout d'abord clair par hypothèse que $-?_1$ est singulière.
  Prenons les classes C_1=$\mathrm{Cl}_J(P_1)$ et C_2=$\mathrm{Cl}_J(P_2)$, qui sont justifiés.
- Prouvons maintenant que pour toute classe C_s de SJ il existe un point P_a de C_s tel que P_a est aussi un point de J et si $\mathrm{H}_{SJ}(C_s)$ est une flèche alors $\mathrm{H}_J(P_a)$ est la même flèche.
  Prenons P un point quelconque de C_s. C'est nécessairement un point de J, car les adresses n'ont pas changé. Si $\mathrm{H}_{SJ}(C_s)$ est une flèche, supposons que $\mathrm{H}_J(P)$ n'est pas identique.
  Nécessairement $S(\mathrm{H}_J(P))$=$\mathrm{H}_{SJ}(C_s)$, donc $\mathrm{H}_J(P)$=$-?_1$ et $\mathrm{H}_{SJ}(C_s)$=$\rightsquigarrow_2$, ce qui fait que P est dans C_1 par singularité de $-?_1$. Prenons alors $P_a = P_2$. Nous avons $\mathrm{H}(P_2)$=$\rightsquigarrow_2$ et P_2 est bien dans C_s dans ce cas.
- Prouvons finalement que SJ vérifie la propriété des classes.
  - Le premier point reste toujours vrai, puisqu'il ne dépend pas des types donnés dans J.
  - Toutes les classes sont justifiées par le même argument que le lemme précédent.
  - Si C_s est telle que $\Phi(C_s)$ est de la forme $([0], \{\not\lambda x.u\})$ avec $x \notin u$ alors il en est de même pour la classe C de P_a dans J. Alors par hypothèse $\mathrm{H}_J(C)$ est un atome b et b est singulier. L'image de cet atome par la substitution est b, donc reste un atome.
    Il est singulier par le même argument que dans la preuve du lemme précédent.
  - Si $\mathrm{H}_{SJ}(C_s)$ est une flèche sous-spécifiée $-?$ alors il en est de même pour $\mathrm{H}_J(P_a)$ et cette flèche est singulière et négative dans J. C'est aussi une flèche négative dans SJ car il

s'agit de la même adresse.
Supposons qu'elle ne soit pas singulière dans SJ. Il existe donc une classe C'_s distincte de C_s telle que $\mathrm{H}_{SJ}(C_s)=\mathrm{H}_{SJ}(C'_s)=-?$. Or il existe P'_a un point de C'_s qui est aussi un point de J, et qui est tel que $\mathrm{H}_J(P'_a) = -?$. Puisque les classes de P_a et P'_a dans SJ sont distinctes (ce sont C_s et C'_s), elles sont distinctes également dans J. Mais la singularité de $\mathrm{H}_J(P_a)$ est alors contredite.
- Si $\mathrm{H}(C_s)$ est une flèche intuitionniste alors il en est de même pour $\mathrm{H}(P_a)$ et cette flèche est positive dans J, donc dans SJ aussi. □

Voici encore le même lemme mais dans le cas de la substitution d'une variable de type par une autre variable de type.

Lemme 3.4.54 *Soit J un ensemble de jugements vérifiant la propriété des classes. Supposons qu'il existe un point $P_1 = (c, x, i)$ tel que $H(P_1)$ est une variable de type α et un point $P_2 = (c, x, j)$ tel que $H(P_2)$ est une autre variable de type β. Soit S la substitution qui transforme α en β. Alors SJ vérifie la propriété des classes.*

Preuve :
Prenons les classes $C_1=\mathrm{Cl}_J(P_1)$ et $C_2=\mathrm{Cl}_J(P_2)$, qui sont justifiés.
- Remarquons juste que ni α ni β ne sont nécessairement singuliers dans J.
- Prouvons que pour toute classe C_s de SJ il existe un point P_a de C_s tel que P_a est aussi un point de J et si $\mathrm{H}_{SJ}(C_s)$ est une flèche alors $\mathrm{H}_J(P_a)$ est la même flèche.
  Prenons P un point quelconque de C_s. C'est nécessairement un point de J, car les adresses n'ont pas changé. Si $\mathrm{H}_{SJ}(C_s)$ est une flèche alors $\mathrm{H}_J(P)$ est la même car les flèches n'ont pas été modifiées, ni la forme des types.
- Prouvons finalement que SJ vérifie la propriété des classes.
  - Le premier point reste toujours vrai, puisqu'il ne dépend pas des types donnés dans J.
  - Toutes les classes sont justifiées par le même argument que le premier lemme de cette série.
  - Si C_s est telle que $\Phi(C_s)$ est de la forme $([0], \{\not\lambda x.u\})$ avec $x \notin u$ alors il en est de même pour la classe C de P_a dans J (qui est contenue dans la première). Alors par hypothèse $\mathrm{H}(C)$ est un atome b et b est singulier dans J. L'image de cet atome par la substitution reste un atome.
    Si il s'agit du même atome alors il est singulier par le même argument que dans la preuve du lemme 3.4.52.
    Si c'est un atome différent, c'est que $b = \alpha$ et que $\mathrm{H}(C_s)=\beta$. Comme α est singulier dans ce cas, nous avons que $C=C_1$. Par suite, C_s contient P_2 et donc sa classe grâce à la substitution. Comme $\Phi(C_s)$ est de la forme $([0], \{\not\lambda x.u\})$ avec $x \notin u$, il en est de même pour $\Phi(C_2)$ et donc β est singulier également dans J.
    Alors β est singulier dans SJ : on a transformé α singulier dans J en β singulier dans J, et la classe C_s contient les deux classes C_1 et C_2.
  - Si $\mathrm{H}(C_s)$ est une flèche sous-spécifiée alors il en est de même pour $\mathrm{H}(P_a)$ et cette flèche est singulière et négative dans J. C'est aussi une flèche négative dans SJ car il s'agit de la même adresse. Et elle est aussi singulière dans SJ par le même argument que précédemment.
  - Si $\mathrm{H}(C_s)$ est une flèche intuitionniste alors il en est de même pour $\mathrm{H}(P_a)$ et cette flèche est positive dans J, donc dans SJ aussi. □

Note 3.4.55 Quand nous considérons un ensemble de jugements J ne contenant qu'un seul terme t, nous notons (c, x, t) et (c, t) les points de J. On remarque que dans ce cas, parler de points est équivalent à parler de classes. Par la suite nous adopterons alors l'utilisation des points.

Un dernier lemme :

Lemme 3.4.56 *Soit t un terme η-long, et considérons un jugement η-long pour t. Prenons J l'ensemble ne contenant que ce jugement. Soit $P = (c, x, t)$ un point tel que $H(P)$ est un atome et P est justifié. Alors les sous-termes justifiants ne sont pas appliqués et ne sont pas de la forme $\lambda\!\!\!^{*}x.u'$.*

Preuve : Nécessairement les sous-termes justifiants ne sont pas appliqués, sinon ils auraient un type flèche. Maintenant si les sous-termes justifiants sont des termes η-longs, comme ils ont un type atomique ils ne s'écrivent pas $\lambda\!\!\!^{*}x.u'$. Dans les autres cas, les sous-termes justifiants ne peuvent jamais être de la forme $\lambda\!\!\!^{*}x.u'$. □

3.4.7 La proposition

Définition 3.4.57 *Nous dirons dans la suite qu'un terme typé (η-long) vérifie la propriété des classes si l'ensemble constitué de ce seul jugement vérifie la propriété des classes.*

Énonçons la proposition importante de cette section :

Proposition 3.4.58 *Soit t un terme typable tel que l'algorithme 3 donne pour t un arbre de typage η-long. Alors le jugement à la racine de l'arbre vérifie la propriété des classes.*

Preuve : par induction sur la complexité de t.

1. $t = \lambda x.t'$: par induction le jugement $\{[x : \tau, x_i : \tau_i \ ; \ y_j : \gamma_j] \vdash t' : \theta\}$ donné par l'algorithme vérifie la propriété des classes. L'algorithme donne alors $\{[x_i : \tau_i \ ; \ y_j : \gamma_j] \vdash \lambda x.t' : \tau \to \theta\}$. On peut voir alors que la propriété des classes est vérifiée.
2. $t = \lambda\!\!\!^{*}x.t'$: idem.
3. $t = (x)\ t_1\ \ldots\ t_n$: considérons J_0 l'ensemble de jugements donnés pour les t_i.
   Alors par induction chacun des jugements de J_0 vérifie la propriété des classes. Nous supposons de plus que chacun des t_i est typé de manière indépendante, c'est à dire avec des variables de type distinctes. On vérifie alors facilement que J_0 vérifie la propriété des classes.
   - La première étape, l'étape d'unification des types des variables, se fait par passage d'ensemble de jugements en ensemble de jugements tels que prévus dans les lemmes 3.4.52, 3.4.53 et 3.4.54. Donc à chaque étape, la propriété est vérifiée. Ainsi à la fin de l'étape d'unification, nous obtenons un ensemble de jugements qui vérifie la propriété des classes et les variables libres communes ont leur type unifié.
     Nous pouvons alors considérer le jugement de $t = (x)\ t_1\ \ldots\ t_n$, où toutes les variables libres ont le type obtenu à cette première étape (si x n'est pas libre, on lui donne comme type une variable de type fraîche).
     Considérons également le jugement de t identique sauf pour le type de la variable x dont le type est celui construit à partir des types des t_i obtenus après unification et utilisant une nouvelle variable de type β.
     Dans les deux jugements le type de t est β. Ces jugement vérifient tous deux la propriété des classes. En effet le type de t étant β, le seul point de la forme (c, t) est $([], t)$. Ensuite pour les points de la forme (c, y, t) avec y différent de x la justification est la même que dans l'ensemble de jugement des t_i. Pour les points de la forme (c, x, t)
     - dans le premier jugement x est traité comme les autres variables.
     - dans le second jugement, le type donné à x est $\tau_1 \rightsquigarrow_1 \ldots \tau_n \rightsquigarrow_n \beta$. Ce type a bien la propriété SPIN grâce à la première étape. Certaines adresses se rapportent à des adresses dans les types τ_i des t_i, d'autres se rapportent aux flèches $\rightsquigarrow_i$ ou à β.

     Dans tous les cas les points sont alors justifiés, et les singularités sont conservées. Les singularités proviennent du fait que la classe d'un point (c, y, t) avec y une variable quelconque est l'ensemble de tous les termes t_i à la fin de la première étape d'unification.

- La seconde étape est l'unification du type de x obtenu à la fin de la première étape avec le type formé par les types des t_i obtenus et β. Prenons comme nouvel ensemble de jugements celui des deux jugements décrits ci-dessus. Cet ensemble vérifie la propriété des classes. On peut alors appliquer les lemmes 3.4.52, 3.4.53 et 3.4.54 pour conclure qu'à la fin nous obtenons deux jugements identiques qui vérifient la propriété des classes. Donc la propriété des classes est vérifiée.

4. $t = (c)\ t_1\ \dots\ t_n$: la première étape (unification des types des variables) se fait comme pour le cas précédent. Pour la seconde étape, nous considérons le terme t avec les types des variables obtenus à l'étape précédente (t étant lui-même de type atomique). La propriété des classes est vérifiée. On unifie ensuite le type de c avec le type formé à partir des types de t_i donnés après la première étape et d'une nouvelle variable de type. Puisque l'algorithme termine avec succès, nous savons que t reste de type atomique et est η-long après l'application de l'unificateur. Supposons que l'unificateur transforme des variables de type apparaissant dans le type des variables en des types flèches. Le lemme précédent nous montre alors que le terme obtenu n'est plus η-long (car les sous-termes justifiants d'un type atomique ne sont ni appliquées ni de la forme $\lambda\!\!\!^{*}x.u$), donc c'est impossible. Ainsi tous les points sont justifiés. Il est clair que les flèches sous-spécifiées, étant singulières avant unification, restent inchangées et singulières. Par conséquent toutes les flèches sont les mêmes et ont même polarité qu'avant l'unification. Les points P pour lesquels le dernier appel à Φ est de la forme $([0], \{\lambda\!\!\!^{*}x.u\})$ avec $x \notin u$ sont tels que $\mathrm{H}(P)$ est un atome a et a est singulier avant unification. Comme l'unification ne touche pas à cet atome, cela reste le cas après unification.

   Donc la propriété des classes est vérifiée.

5. $t = (\lambda\!\!\!^{*}x_1 \dots x_m(\dots)\dots)\ t_1\ \dots\ t_m$: tout se fait comme pour le cas précédent. Nous remarquons seulement qu'il y a $m+1$ jugements dans l'ensemble de la première étape (nous ajoutons en effet le terme $\lambda\!\!\!^{*}x_1 \dots x_m(\dots)\dots$). □

Nous terminons enfin par le corollaire suivant, qui permet d'avoir des types principaux dans le système de typage initial, sans flèches sous-spécifiées.

Corollaire 3.4.59 *Soit t un terme typable η-long de type T. Supposons qu'il existe une adresse c telle que $f(c,T)$ a pour tête $\to$ et que c est négative. Alors t est typable (η-long) de type T' où T' est T où l'on a remplacé la flèche $\to$ en $\multimap$. Ainsi on peut donner pour type principal à tout terme η-long le type donné par l'algorithme de typage, dans lequel toutes les flèches sous-spécifiées sont changées en $\to$.*

Preuve : Comme t est typable η-long, son type principal TP est η-long. Il existe une substitution S telle que $STP{=}T$. Par la proposition précédente, TP n'a pas de flèche $\to$ négative. Donc S transforme une flèche sous-spécifiée $-?_1$ en $\to$, ou une variable de type α en un type A contenant la flèche $\to$.
Dans le premier cas, comme $-?_1$ est unique, nous pouvons changer S en S' qui est S avec l'image de $-?_1$ est $\multimap$. Alors $S'TP = T'$ et t est bien typable de type T'.
Dans le second cas, prouvons que α est unique. Dans TP, il y a une adresse c pour α. Le point P est justifié par la proposition précédente et les termes du dernier appel sont η-long. Les sous-termes justifiants sont de type α. L'adresse du dernier appel n'est pas $[]$, car alors les sous-termes justifiants sont aussi η-longs, or avec le type T, ces termes ont un type flèche. Donc, les sous-termes justifiants sont soit des $(x)\ t_1\ \dots\ t_k$ soit des variables. Ce ne peut pas être $(x)\ t_1\ \dots\ t_k$, car avec le type principal, ils ont pour type α et donc ne sont pas appliqués. Or pour T ils ont un type flèche et t est η-long. Donc il s'agit de variables, le dernier appel étant $([0], \{\lambda\!\!\!^{*}x.u\})$. Nécessairement les variables x n'apparaissent pas dans les termes u, car sinon elles ne seraient pas appliquées, mais pour T elles ont un type flèche. La proposition précédente nous assure alors que α est bien singulier, donc unique puisque nous ne parlons que d'un terme.

Comme α est unique, nous pouvons prendre S' qui est S pour laquelle α est transformé en A', qui est A où la flèche $\to$ a été transformée en $\multimap$. Alors $S'TP{=}T'$, et t est bien typable de type T'. □

3.5 Le lien avec les ACGs

Dans ce chapitre nous avons décrit un λ-calcul, sur lequel nous avons donné des propriétés sur le type principal des termes de certains fragments. Mais qu'en est-il des ACGs, dans le cadre desquelles a été défini ce λ-calcul ?
Le lien est le suivant : Comme nous l'avons déjà signalé, les ACGs ont été développées à l'origine sur le λ-calcul linéaire, et le manque d'expressivité pour la sémantique a motivé l'extension de ce calcul vers un calcul avec deux types de variables, puis deux flèches. Si la question du type principal est naturelle et intéressante en soi, elle est en fait posée lors de l'utilisation des ACGs.
En effet, l'utilisation des ACGs est la suivante. Deux signatures, la signature abstraite et la signature objet, sont définies (voir le chapitre 1, section 1.4.1 pour plus d'informations). C'est à dire que l'on donne les constantes de chaque signature, et le type de celles-ci. Ensuite on définit le lexique. Pour donner toutes les informations, il faut donner l'image des constantes de la signature abstraite, ainsi que l'image des types atomiques.
On doit alors ensuite vérifier que le morphisme vérifie sa propriété de commutation, à savoir que l'image de tout terme t est typable de type l'image du type de t. Pour faire cela, on doit pouvoir déterminer un type principal du terme image, et vérifier que ce type est plus général que l'image du type de t.

Il y a en fait une utilisation plus intéressante pour l'utilisateur, qui est de ne donner que l'image des constantes, mais pas des types atomiques, la machine ayant pour but de trouver grâce à la propriété de commutation le lexique complet. Le problème revient alors à unifier les types qui doivent être l'image des types atomiques.
Soyons plus précis : Notons c_i les constantes de la signature abstraite, et t_i leur image par le lexique. Notons ensuite T_i le type des constantes (qui ne contient pas de variable de type), et P_i le type principal des t_i (qui contient donc des variables de type). Nous devons avoir le fait que l'image de T_i est un cas particuliers de P_i.
L'image $T' = \mathcal{L}(T)$ d'un type T est la même que T, en renommant seulement les types atomiques A en A' par exemple. Il suffit alors de trouver la valeur des A', qui sont les images des types atomiques A.
On obtient un ensemble d'égalités $T'_i = P_i$ que l'on doit unifier. Les variables de type se trouvent à la fois dans T'_i et dans P_i.

Les termes η-long ont aussi une existence au sein des ACGs, lorsqu'il s'agit de faire l'analyse syntaxique. La question est la suivante : étant donné un terme objet u, quels sont les termes abstraits t de type S tels que $\mathcal{L}(t) = u$?
Dans un cadre plus général, étant donnés Γ un contexte abstrait, v un terme objet η-long et β-normal, et α un type abstrait, chercher un terme abstrait t tel que $\Gamma \vdash t : \alpha$ et $\mathcal{L}(t) =_{\beta\eta} u$.
Le problème revient à résoudre d'étape en étape des problèmes de filtrage (c'est à dire d'unification d'équations avec terme de droite sans variable). Ce problème est NP-complet, et un semi-algorithme du λ-calcul donné par Huet (cf [Hu]) utilise les formes η-longues des termes. C'est pourquoi le terme objet est considéré η-long.

Chapitre 4

Un démonstrateur générique

Afin d'être capable de justifier les règles décrites à l'aide du langage restreint, il a fallu implémenter un démonstrateur conçu en fonction des spécificités du problème. Les premières tentatives ont été faites en utilisant la tactique trivial de l'assistant de preuve PhoX développé par Christophe Raffalli.

Cependant la tactique trivial de PhoX n'a pas pour vocation de prouver les buts provenant de l'interprétation du langage restreint. Il a pu être observé que malgré l'évidence de certaines étapes de démonstrations, PhoX se trouvait incapable de les justifier.

Deux choses pouvaient être mises en cause. La première est la formule qui était donnée à prouver. Dans certains cas celle-ci pouvait être trop complexe, et être simplifiée. C'est une des choses qui a été faites durant ce travail sur le projet (voir le chapitre 2).

La seconde chose est la tactique utilisée par le démonstrateur. La tactique trivial de PhoX est basée sur l'utilisation de règles de la déduction naturelle, et nécessite entre autres de retourner en arrière (le *back-tracking*) pour trouver des preuves lorsque une branche explorée s'avère mauvaise. La volonté d'éviter le *back-tracking* et aussi de partager au maximum les connaissances acquises lors de la preuve ont motivé l'utilisation de la méthode de la résolution.

D'autre part nous sommes partis de l'idée que les étapes de preuves faites en langue naturelle ne sont généralement que de petites étapes. En effet, la preuve étant faite pour que le lecteur puisse la comprendre, on peut estimer que pour les comprendre il n'y a pas beaucoup de raisonnement à faire d'une étape à l'autre. En notant bien entendu que ceci varie selon le niveau de la personne qui fait la preuve et du lecteur.

En d'autres termes la justification de chaque étape ne nécessite généralement pas d'entrer profondément dans les formules. Ceci a mené à l'idée d'utiliser non pas des clauses contenant seulement des formules atomiques comme c'est le cas lors de la résolution standard mais contenant des formules quelconques. Ainsi la mise sous forme clausale n'est pas préliminaire à la recherche de la preuve, mais se fait durant celle-ci.

Ensuite, le fait que dans une preuve on donne des justifications afin que le lecteur en comprenne chaque étape entraîne que le démonstrateur doit être capable de suivre les indications apportées par l'utilisateur. L'utilisation de poids et de contraintes peut permettre cela.

Enfin nous sommes arrivés à la notion de démonstrateur générique, c'est à dire indépendant de la logique. Cela vient de la volonté d'implémenter un système modulaire permettant de changer avec aisance la logique utilisée dans le démonstrateur sans avoir à modifier les fonctions internes du démonstrateur. Cela a permis de tester rapidement le démonstrateur sur des logiques simples comme la logique propositionnelle, avant de passer au premier ordre, puis à la logique d'ordre supérieur utilisée dans PhoX.

Ce chapitre détaille les idées développées pour l'implémentation du démonstrateur, qui a été faite dans le langage Objective Caml.

Dans une première partie nous en donnerons les idées principales. Puis dans une seconde section nous parlerons des bases sur lesquelles il repose, à savoir la résolution et les règles de décomposition. Nous traiterons ensuite des stratégies appliquées pour augmenter l'efficacité de la résolution. Nous finirons par une section parlant des logiques utilisées par le démonstrateur

jusqu'à ce jour, où seront données encore quelques heuristiques ayant pour but de rendre les preuves rapides.
D'autres informations sur le démonstrateur seront trouvées dans l'annexe III dans laquelle son fonctionnement est plus détaillé.

4.1 Ce que l'on souhaite

Il est bien certain que la volonté n'est pas de développer un démonstrateur susceptible de prouver n'importe quelle formule valide. En effet, le démonstrateur a pour but de justifier des étapes de preuves et certainement pas de faire des preuves de théorèmes. Par conséquent il faut chercher à faire un démonstrateur capable de suivre des indications que l'on peut lui donner, et répondre le plus rapidement possible dans un cas positif comme dans un cas négatif.
Pour répondre vite dans un cas positif, il faut effectivement que le démonstrateur parvienne à suivre les indications données par l'utilisateur. Celles-ci peuvent être données par différents moyens :

- le poids donné à des hypothèses peut varier selon que l'on considère que ce sont des hypothèses importantes (dans ce cas on donne un poids faible) ou inutiles (poids élevé). Leur poids favorise ou défavorise leur utilisation, et ainsi plus le poids des bonnes hypothèses est faible, plus vite la preuve peut être faite.
- l'utilisation de certaines hypothèses par d'autres, ou l'application de formules quantifiées avec certaines valeurs particulières peuvent être favorisées grâce à des contraintes.

Pour répondre vite dans un cas négatif, la solution adoptée est de limiter le temps de recherche de la preuve, et de répondre que la preuve n'a pas été trouvée, ce qui peut permettre à l'utilisateur de détailler plus sa preuve, soit en ajoutant des étapes, soit en précisant ce qu'il utilise comme hypothèse.

4.1.1 Des preuves en surface

Il est entendu que l'humain n'est pas en mesure d'avoir en tête à tout moment la complexité de toutes les hypothèses qu'il a en sa possession lorsqu'il fait une preuve. Il se concentre sur une partie seulement des hypothèses à chaque étape, et de plus n'en utilise pas tous les détails. On peut alors se dire que pour valider une étape de démonstration, le démonstrateur doit pénaliser l'utilisation de sous-formules trop profondes dans les hypothèses, et favoriser une preuve courte restant en surface des hypothèses.
Nous avons choisi comme stratégie de preuve la résolution (détaillée à la section 4.2.1). Lorsque l'on souhaite prouver une formule par résolution, il faut extraire de cette formule un ensemble de clauses contradictoire. Généralement, cette extraction se fait avant l'application de la stratégie de résolution. Mais dans notre cas, il a fallu implémenter un démonstrateur qui forme cet ensemble de clauses dans le cours de la preuve, et non au préalable.
Cela signifie que les clauses manipulées par le démonstrateur ne sont plus des ensembles de littéraux atomiques (formules atomiques ou négations de formules atomiques), mais des ensembles de formules quelconques. Celles-ci sont alors décomposées, et nous verrons que décomposer les formules peut être vu comme utiliser des clauses de règle à la section 4.2.2.

4.1.2 L'indépendance à la logique

L'idée de faire un démonstrateur indépendant de la logique provient de la volonté de ne pas avoir à ré-implémenter un démonstrateur pour chaque nouvelle implémentation d'un assistant de preuve pour lequel le système logique peut évoluer. Cela permet donc une souplesse d'utilisation.
En contre partie, il s'agit alors de demander un certain nombre de pré-requis à la logique pour pouvoir l'utiliser et obtenir un démonstrateur. Voir la section 4.4.1 pour plus de précisions.
Principalement, la logique est le module qui manipule les formules. Elle définit donc ce qu'est une formule, a un algorithme d'unification, et définit également ce que sont les contraintes.

Ces dernières peuvent être de tout ordre. Elles peuvent par exemple être là pour permettre de détecter l'utilisation d'une hypothèse par une autre ou empêcher l'unification de certains littéraux ne vérifiant pas certaines conditions.

4.2 Les bases

Nous redéfinissons ici les bases sur lesquelles repose le démonstrateur, à savoir le principe de résolution.

4.2.1 La résolution

Les ouvrages traitant de la démonstration automatique tels que [DaNoRa], [ChaLe] et [BaGa] détaillent la méthode de la résolution. L'idée principale est de transformer la formule à prouver en un ensemble de formules atomiques qui est contradictoire si et seulement la formule est vraie. Rappelons ici le principe, dans le cas du premier ordre. Le lecteur pourra se reporter à l'annexe I pour voir la définition des formules du premier ordre, ainsi que l'unification.

Définition 4.2.1 *Un littéral est une formule atomique A ou la négation d'une formule atomique $\neg A$.*

Définition 4.2.2 *Une clause est un ensemble de littéraux. Nous noterons C_1=L_1,…,L_n une clause. La clause vide, ne contenant aucun littéral, est notée $\Box$.*

Une clause représente la clôture universelle de la disjonction de ses éléments. Chaque clause est donc une formule close. Un ensemble de clauses représente la conjonction des clauses.
Le principe de la résolution repose sur deux règles. La première est la règle de résolution :

$$\frac{C_1, L_1 \quad C_2, \neg L_2 \quad \sigma = mgu(L_1, L_2)}{C_1\sigma, C_2\sigma}\, res$$

Où *mgu* représente l'unificateur le plus général. Il est à noter que du fait que les clauses sont vues comme des formules closes on considère toujours que les variables des clauses sont distinctes. C'est à dire que si $L_1 = A(x)$ et $L_2 = A(f(x))$, on renomme les variables et on unifie $R(x)$ et $R(f(y))$. La règle de résolution est une règle de logique classique, puisqu'elle s'appuie sur le tiers exclu $(A \vee \neg A)$. Elle se comprend de la manière suivante, dans le cas propositionnel pour simplifier : Si l'on a $A \vee B_1$ et $\neg A \vee B_2$, si l'on admet que l'on a soit A soit $\neg A$ alors on a bien $B_2 \vee B_1$. On appelle aussi cette règle résolution binaire.

La seconde règle est la règle de contraction :

$$\frac{C_1, L_1, L_2 \quad \sigma = mgu(L_1, L_2)}{C_1\sigma, L_1\sigma}\, contr$$

Cette règle se comprend fort aisément, puisque il s'agit d'éliminer les redondances : Si l'on a $A \vee A \vee B$, alors on a $A \vee B$.

Définition 4.2.3 *Un ensemble E de clauses est dit incohérent si l'on peut dériver la clause vide à partir de E.*

Définition 4.2.4 *Un ensemble E de clauses est dit contradictoire si il n'est vrai dans aucune interprétation. Autrement dit dans toute interprétation il existe une clause qui est fausse.*

Le but de la résolution est, à partir d'un ensemble de clauses contradictoire, d'aboutir en utilisant les règles de résolution et de contraction à la clause vide $\Box$, qui représente l'absurde (En effet la règle de résolution quand C_1 et C_2 sont vides dit que si on a A et $\neg A$, ce qui est absurde, alors on a la clause vide).
Dans le cas du premier ordre, on a le théorème suivant :

Théorème 4.2.5 *Un ensemble de clauses est contradictoire si et seulement si il est incohérent.*

A partir de là, lorsque l'on souhaite prouver une formule F en utilisant cette méthode, on transforme la formule $\neg F$ (dont on veut prouver qu'elle est contradictoire) en un ensemble de clauses dont on sait qu'il est contradictoire si et seulement si $\neg F$ est contradictoire. Il y a plusieurs méthodes pour faire cela, que le lecteur intéressé pourra trouver dans [DaNoRa] par exemple. Nous ne les redonnons pas ici car nous introduisons une autre manière d'appliquer la résolution, qui est de décomposer pendant la résolution.

4.2.2 Les règles de décomposition

Si le démonstrateur utilise des clauses qui sont des ensembles de formules non nécessairement atomiques (que nous appelerons encore littéraux pour qu'une clause reste un ensemble de littéraux), il faut intégrer au démonstrateur le moyen de décomposer ces formules.
Il suffit en fait d'appliquer les règles usuelles qui permettent d'obtenir un ensemble de clauses à partir d'une formule. Par exemple, supposons que nous ayons la clause $C = F, C'$, avec F une formule et C' le reste de la clause. Si F est de la forme $A \to B$, alors on peut transformer C en la clause $\neg A, B, C'$. En effet la formule $A \to B$ est équivalente à $\neg A \vee B$ et comme une clause est interprétée par la disjonction de ses formules, on peut éliminer la disjonction.
Si F est de la forme $A \wedge B$, alors la clause peut être décomposée en deux clauses, A, C' et B, C'. Ici nous utilisons la distributivité de la conjonction sur la disjonction : $(A \wedge B) \vee C'$ est équivalent à $(A \vee C') \wedge (B \vee C')$. Un ensemble de clauses étant interprété par la conjonction des clauses, cela nous permet d'ajouter alors les deux clauses.
La question est alors de savoir comment permettre cette décomposition dans le démonstrateur qui, rappelons-le, ignore la forme des formules qu'il traite.
Décomposer, c'est appliquer une fonction de la logique qui prend une formule, qui regarde la tête de la formule et, connaissant sa polarité, donne les sous-formules en lesquelles elle se décompose. Cette fonction donne donc les règles de décomposition des formules qui lui sont données.
Nous verrons dans le chapitre 5 une représentation des règles de décomposition, dans le cas du premier ordre, par des règles logiques. Nous verrons également dans ce même chapitre le résultat dont nous parlions à la section précédente, à savoir que l'on conserve en appliquant les règles un ensemble contradictoire, et que le système est complet avec ces décompositions.

Remarque 4.2.6 Il est possible de voir ces règles de décomposition comme l'application d'une résolution d'ordre supérieur, la clause permettant la décomposition étant alors appelée une clause de règle.

Exemple 4.2.7 Pour $F = A \to B$, la clause qui nous permet d'obtenir la décomposition est la suivante : $\neg(X_1 \to X_2), \neg X_1, X_2$, avec X_1 et X_2 des variables d'ordre supérieur. On considère évidemment que la résolution avec cette clause ne se fait que sur le littéral $\neg(X_1 \to X_2)$. Ainsi F s'unifie bien avec ce littéral, et la clause résultante est $\neg A, B, C'$ où C' est le reste de la clause dans laquelle se trouve F.

Exemple 4.2.8 Pour $F = A \wedge B$, il y a deux clauses de règle : $\neg(X_1 \wedge X_2), X_1$ et $\neg(X_1 \wedge X_2), X_2$ qui nous permettent d'obtenir A, C' et B, C'. Dans ces deux clauses de règle n'est autorisé de résolution qu'avec le seul le littéral $\neg(X_1 \wedge X_2)$, ce qui permet effectivement de décomposer.

Nous avons vu dans ces deux exemples que la résolution avec les clauses de règles est restreinte, c'est à dire que certains littéraux ne doivent pas subir la résolution. Seulement certains littéraux (dans nos exemples seulement un littéral à chaque règle) appelés littéraux principaux peuvent subir une règle de résolution.
Nous pouvons remarquer que les clauses de règles sont en fait des tautologies, car la formule qui les représente est vraie : $\neg(X_1 \to X_2) \vee \neg X_1 \vee X_2$, $\neg(X_1 \wedge X_2) \vee X_1$ et $\neg(X_1 \wedge X_2) \vee X_2$ sont des formules valides. Il semble donc naturel de devoir restreindre leur utilisation puisque la

méthode de résolution a pour but de trouver une contradiction et qu'une tautologie n'apporte pas par elle-même de contradiction.
Une autre raison à cette restriction est que les variables d'ordre supérieur pouvant être unifiées par n'importe quelle formule, il devient possible d'ajouter des formules plus complexes avec de nouvelles variables. Par exemple pour la clause de règle $\neg(X_1 \wedge X_2), X_1$, nous pourrions faire la résolution sur le littéral X_1 avec un littéral $\neg F$, ce qui donnerait alors la clause $\neg(F \wedge X_2)$. Or nous souhaitons éviter d'ajouter des formules plus complexes et des variables d'ordre supérieur, qui sont inutiles et augmenteraient dramatiquement le nombre de clauses possibles. La présentation en clause de règle étant assez peu précise, et nécessitant ces restrictions, il est tout de même préférable de voir la décomposition comme décrite plus haut.

4.3 Les stratégies

Certaines stratégies peuvent être utilisées afin d'améliorer la recherche de preuve. Il est possible tout en gardant la complétude de réduire le nombre de clauses formées ou gardées en mémoire. Il y a deux types de stratégies :
– les stratégies de restriction des résolutions qui permettent de réduire le parcours des arbres de recherche.
– les stratégies d'élimination qui permettent de réduire l'ensemble des clauses produites.
Le but de ces stratégies est bien entendu de faire gagner mémoire et temps à la machine.

Remarque 4.3.1 Nous ne parlerons que de quelques stratégies, notre volonté n'étant pas d'être exhaustif. Toutes les stratégies discutées ici, sauf mention contraire, ont été effectivement implémentées dans le démonstrateur.

4.3.1 Les stratégies standards

Les résolutions positives et négatives sont des stratégies de restriction, qui sont des cas particuliers de la résolution sémantique. Les deux stratégies d'élimination les plus communes sont l'élimination des tautologies et l'élimination des clauses subsumées.
Ces stratégies étant comme le titre de la section le précise standard, il ne sera pas fait de preuves de la complétude de celles-ci, mais il sera seulement donné une idée intuitive simple de la raison de ces stratégies, et uniquement de la raison de celles-ci. On pourra voir [ChaLe], [Du], [Fi], [Le1] ou [Ro] pour des preuves plus complètes.

Résolution sémantique

Définition 4.3.2 (résolution sémantique) *Étant donnée une interprétation $\mathcal{I}$, la résolution sémantique consiste à ne faire la résolution entre deux clauses que si l'une d'elle est fausse dans l'interprétation.*

Définition 4.3.3 (résolution positive et négative) *Il y a deux cas particuliers de la résolution sémantique :*
– La résolution négative consiste à prendre l'interprétation dans laquelle les atomes sont vrais ;
– La résolution positive consiste à prendre l'interprétation dans laquelle les atomes sont faux.

Remarque 4.3.4 Les noms donnés à la résolution négative et positive peuvent paraître étrange compte tenu de leur définition, mais il s'agit en fait dans le cas négatif (resp. positif) de ne faire une résolution que si l'une des clauses ne contient que des littéraux négatifs (resp. positifs) puisque seules celles-ci sont fausses dans le modèle choisi.

Note 4.3.5 Dans notre démonstrateur, c'est la résolution positive ou négative qui est implémentée, en option de surcroit. Il existe une stratégie, l'hyper-résolution, qui consiste à faire en une seule étape (ou règle) la résolution entre la clause fausse C et un ensemble de clauses de taille le nombre de littéraux de C. La règle appliquée est alors la suivante pour le cas négatif :

$$\frac{\neg A_1, \ldots \neg A_n \quad A'_1, C_1 \quad \ldots \quad A'_n, C_n}{C_1\sigma, \ldots, C_n\sigma}$$

où σ est l'unificateur simultané de l'ensemble $\{A_i \sim A'_i\}$

Proposition 4.3.6 *La résolution sémantique est complète.*

Preuve (idée) : Comme la résolution cherche à trouver une contradiction, il est inutile de chercher une contradiction avec des clauses que l'on sait vraies dans un modèle. Donc on fait des résolutions avec au moins une des clauses que l'on sait fausse dans le modèle choisi. □

Élimination des tautologies

Définition 4.3.7 *Une clause est une tautologie si elle contient une formule F et sa négation $\neg F$*

Proposition 4.3.8 *La stratégie qui consiste à supprimer les tautologies et à ne pas en former est complète.*

Preuve (idée) : Puisqu'une clause représente la disjonction de ses formules, il est clair que grâce au tiers exclus une tautologie est bien une tautologie, c'est à dire une formule vraie dans tout modèle. Or la résolution est une méthode qui a pour but de trouver une contradiction. Ainsi, si S est un ensemble de clauses, et C est une tautologie, S est contradictoire si et seulement si $S \cup \{C\}$ est contradictoire. On peut donc éliminer les tautologies. □

Remarque 4.3.9 Attention toutefois au fait que cette stratégie peut rendre incomplète une stratégie complète. C'est à dire qu'il existe des stratégies complètes qui ne peuvent pas être étendue à l'élimination des tautologies en gardant la complétude. Un exemple est la *lock resolution* (c.f. [Le1]), qui est une stratégie utilisant des ordres dans les clauses. Elle peut cependant être ajoutée à la stratégie l'élimination des clauses subsumées.

Note 4.3.10 Comme nous l'avons vu précédemment, le démonstrateur utilise des clauses de règle, qui sont des clauses représentant des formules toujours valides. Or la présence de ces clauses est nécessaire à la complétude du système car elles permettent la décomposition. Il ne s'agit cependant pas de clauses tautologiques vérifiant la définition donnée plus haut.
Le problème de chercher une contradiction sans utiliser des formules que l'on sait valides nous impose cependant à voir les clauses de règle plus comme des outils permettant la décomposition que comme de véritables clauses. Le fait d'en avoir restreint l'usage uniquement pour décomposer est d'ailleurs une raison pour les voir comme des outils.

Élimination des clauses subsumées

Définition 4.3.11 *Une clause C subsume une clause C' s'il existe une substitution σ telle que $C\sigma \subset C'$. On dit aussi que C' est subsumée par C.*

Proposition 4.3.12 *La stratégie qui consiste à supprimer toute clause C' subsumée par une autre clause C sans que la substitution qui vérifie $C\sigma \subset C'$ unifie des littéraux de C est complète.*

Preuve (idée) : La condition sur la substitution est nécessaire, car sinon toutes les clauses obtenues par une contraction pourraient être éliminées. En effet, prenons la clause $C = A(x), A(y), B$ et $C' = A(x), B$ sa contractée. Si $\sigma = \{y/x\}$ alors $C\sigma = C'$. Ainsi la clause contractée C' est subsumée par la clause C. Si l'on pouvait éliminer toutes les clauses subsumées alors cela signifierait que la résolution sans la contraction serait complète, ce qui n'est pas le cas. En effet un ensemble contradictoire de clauses à deux littéraux (il en existe) ne peut avoir de preuve par résolution pure car toute clause obtenue aurait deux littéraux. Ceci étant

dit, C implique $C\sigma$ qui n'est qu'un cas particulier de C, et si $C\sigma \subset C'$, alors $C\sigma$ implique C'. Donc si S est un ensemble de clauses, $S \cup \{C, C'\}$ est contradictoire si et seulement si $S \cup \{C\}$ est contradictoire. □

Remarque 4.3.13 Il est à noter qu'il y a deux types de subsomption : La clause C' peut apparaître après C dans le processus de la résolution, c'est la subsomption avant, ou elle peut apparaître avant, c'est la subsomption arrière.

4.3.2 La séparation sans séparation

Une des difficultés rencontrées a été la preuve des équivalences, pour lesquelles un humain fait la distinction de deux cas, prouvant une implication puis la réciproque, de manière indépendante. La méthode de résolution a l'avantage de n'avoir pas besoin de retour arrière dans les preuves, car elle partage toujours les connaissances acquises en cours de preuve.
Cependant ce partage s'avère mauvais lorsque l'on cherche à prouver des choses assez fortement indépendantes. Par exemple les preuves d'équivalence, et les preuves par cas (si l'on a une hypothèse du type $A \vee B$).
Si l'on n'adopte aucune stratégie, de telles preuves peuvent s'avérer catastrophiques tant par le temps pris pour les obtenir que par la taille des preuves elles-mêmes, qui peuvent mélanger les cas entre eux dans une preuve incompréhensible.
Il est possible d'améliorer l'efficacité en coupant l'ensemble de clauses en 2. Par exemple, si S est un ensemble de clauses et $C = A \vee B$ est une clause telle que les variables libres de A et de B sont distinctes alors $S \cup \{C\}$ est contradictoire si et seulement si $S \cup \{A\}$ et $S \cup \{B\}$ sont contradictoires.
L'ennui avec cette méthode de séparation est justement qu'elle coupe l'ensemble de clauses. Cela signifie que l'on perd le partage des connaissances, et des choses qui pourraient être communes, ne dépendant pas du cas dans lequel on se place, sont prouvées dans le premier ensemble, puis une autre fois dans le second.
Il existe heureusement un moyen de remédier à cela. Il s'agit de la séparation sans séparation (*splitting sans splitting* en anglais) qui comme son nom l'indique permet de séparer des clauses, sans pour autant faire deux ensembles de clauses.
L'idée est d'ajouter de nouvelles variables propositionnelles, appelées littéraux de séparation. S'il existe une clause $C = A \vee B$ telle que les variables libres de A et de B sont distintes, alors on crée un nouveau littéral de séparation P, et on ajoute les clauses A, P et $B, \neg P$.
Les littéraux de séparations sont en fait les représentants de littéraux et par conséquent il est possible d'éviter de donner à chaque fois un nom nouveau. Le littéral P ci-dessus représente ainsi le littéral B.

Exemple 4.3.14 Si nous disposons des clauses A, B et A, C, nous pouvons couper la première en $A, \neg p$ et $p, \neg B$, le nouveau littéral p représentant A. Alors la seconde clause peut être coupée en $A, \neg p$ et p, C. Ainsi on conserve une certaine symétrie en rapport avec A pour B et C.

Le fait de réutiliser un littéral de splitting a quelques avantages
- cela permet dans certains cas de réduire le nombre de clauses. Dans l'exemple ci-dessus nous avons obtenu deux fois $A, \neg p$, alors que si nous avions utilisé un nouveau littéral de séparation nous aurions eu une clause supplémentaire.
- cela permet de ne pas faire trop augmenter le nombre de nouveaux symboles, qui font augmenter le nombre de clauses possibles de manière importante.

Remarque 4.3.15 Il vaut mieux éviter de faire la résolution sur les littéraux de séparation, sous peine de retrouver la clause initiale, ce qui est une perte de temps.

Il vaut mieux faire les résolutions sur les littéraux initiaux, pour aboutir à des clauses ne contenant que des littéraux de séparation, que nous appellerons des clauses de séparation.

Sur l'ensemble de clauses de séparation, qui se crée au fur et à mesure de la preuve, on cherche alors à déduire par résolution la clause vide.

Le lecteur intéressé pourra lire par exemple [Ro] ou [RiVo] où est discuté de cette stratégie de résolution.

4.3.3 La résolution linéaire

Cette stratégie, et plus précisément la déduction linéaire ordonnée (OL-déduction) a été utilisée pour traiter le cas les clauses de séparation, c'est à dire l'ensemble des clauses ne contenant que des littéraux de séparation. En effet, à chaque fois qu'une clause de séparation est ajoutée, on teste si celle-ci est contradictoire avec les précédentes, en mettant à jour un arbre binaire représentant toutes les interprétations de l'ensemble des clauses de variables propositionnelles que sont les littéraux de séparation (cf. annexe II pour plus de précision).

Si la nouvelle clause ajoutée rend l'ensemble contradictoire, c'est à dire si toutes les interprétations représentées dans l'arbre rend l'ensemble de clauses contradictoire, alors la résolution linéaire ordonnée est une méthode qui permet de trouver une preuve par résolution de manière efficace.

Notons bien que si l'on sait que l'ensemble est contradictoire, c'est que nous venons d'en établir une preuve. On peut se demander pourquoi on cherche alors une preuve supplémentaire. Il s'agit en fait d'afficher une preuve de ce fait :

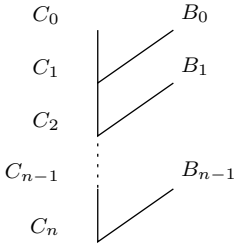
- la première preuve n'est pas satisfaisante pour quelqu'un souhaitant la vérifier, car aucun affichage élégant ne pourrait en être fait. Donner un ensemble contradictoire sans expliquer clairement en quoi il est contradictoire peut rendre la vérification fastidieuse.
- Par souci d'homogénéité nous souhaitons faire une preuve par résolution.

Dans [ChaLe], cette méthode de résolution linéaire ordonnée est définie et détaillée. A l'aide d'exemples tirés de cette source nous rappelons ici le principe de la méthode.

Définition et exemples

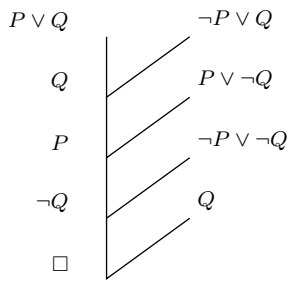
Définition 4.3.16 *Soit S un ensemble de clauses et C_0 une clause de S. Une déduction linéaire de C_n par S avec C_0 comme clause de départ est la déduction de la forme indiquée dans la figure qui suit, où :*

Pour $i = 0, 1, \ldots, n-1$, C_{i+1} est un résolvant de C_i (appelée clause centrale) et B_i (appelée clause de coté). De plus, chaque B_i est soit dans S, soit est un C_j pour un j, $j < i$.



Exemple 4.3.17

Considérons $S = \{P \vee Q, \neg P \vee Q, P \vee \neg Q, \neg P \vee \neg Q\}$. Voici la déduction linéaire de $\square$ par S avec $P \vee Q$ comme clause de départ. Il y a quatre clauses de coté, dont une provient des clauses centrales (il s'agit de Q).



En partant d'un ensemble S de clauses et d'une clause C de S vérifiant : S est incohérent et $S - \{C\}$ est cohérent, il existe une déduction linéaire de la clause vide $\square$ par S avec comme clause de départ C. Mais il est parfois obligatoire de prendre une clause centrale lors de la dérivation, ce qui peut engendrer un grand nombre de possibilités.
Le but de la méthode que nous allons voir ici est justement de limiter l'utilisation des clauses centrales lors de la déduction linéaire. Pour cela, nous considérerons maintenant que l'écriture d'une clause est ordonnée, et parlerons de clause ordonnée. Le principe est qu'une clause de départ ou centrale ne peut être résolue que si le littéral résolu est celui à droite de la clause. De plus, nous allons ajouter une autre idée, qui est l'information sur les littéraux résolus. Nous allons en effet garder et encadrer dans la clause conclusion d'une résolution l'un des littéraux résolus.

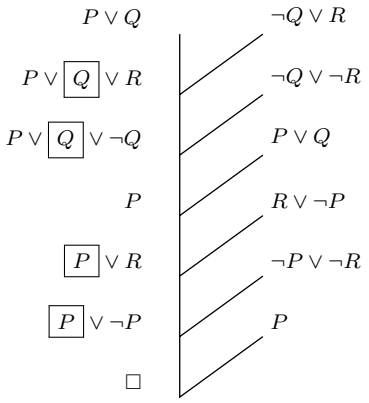
Exemple 4.3.18

Si la clause centrale ou de tête est $P \vee Q$, seul le littéral Q peut être résolu. Supposons qu'au cours de la résolution linéaire on puisse utiliser $P \vee \neg Q$ comme clause de coté. Alors nous pouvons faire une résolution, et le résultat sera (nous gardons Q comme information) $P \vee \boxed{\text{Q}}$. Nous ajoutons la règle suivante : si un littéral encadré est à droite de la clause, il est éliminé. Ainsi, dans l'exemple ci-dessus, $\boxed{\text{Q}}$ est éliminé, et il ne reste que P.
Quand dans une clause ainsi formée, il co-existe un littéral encadré et son opposé (non encadré) à droite de la clause, alors seulement on a le droit d'utiliser une clause centrale. En fait il est inutile de faire une résolution et il suffit d'utiliser la règle suivante : on forme la clause centrale suivante en éliminant le littéral de droite (réduit). On ré-applique ensuite la règle d'élimination des clauses encadrées à droite si nécessaire.
On ajoute une dernière règle : si dans une clause ordonnée apparaît plusieurs fois le même littéral, on supprime ceux qui sont le plus à droite, pour ne laisser que celui qui se trouve le plus à gauche.

Exemple 4.3.19 Un dessin valant mieux qu'un long discours, voici un exemple. Soit

$$S = \{P \vee Q, \neg Q \vee R, R \vee \neg P, \neg Q \vee \neg R, \neg P \vee \neg R\}$$

Voici une OL-déduction (OL pour Ordered Linear) de la clause vide par S avec $P \vee Q$ comme clause de départ.



On voit bien dans cet exemple que les troisième et sixième réductions sont inutiles lorsque l'on suit la règle d'élimination du littéral et de son opposé encadré. On peut donc supprimer les deux branches de résolution correspondant à l'usage de cette règle.

Cette méthode est correcte.

Théorème 4.3.20 (correction de l'OL-déduction) *Si C est une clause ordonnée de l'ensemble contradictoire S telle que $S - \{C\}$ soit non contradictoire alors il existe une OL-réfutation par S avec C comme clause de départ.*

4.4 Les logiques utilisées

Comme nous l'avons dit dans la section 4.1.2, le démonstrateur est générique, il ne connait pas la structure des formules qu'il manipule. Il fait appel à un module de logique qui définit le type des formules, contient un algorithme d'unification et donne les décompositions. Nous donnerons plus précisément ce qui est requis par le module de logique.
Le démonstrateur a été jusqu'à présent utilisé par la logique propositionnelle et la logique du premier ordre. Une version de la logique du premier ordre avec le langage restreint a également été développée, ce qui a permis d'utiliser la notion de contraintes. Récemment le démonstrateur a été intégré à l'assistant de démonstration PhoX, basé sur une logique d'ordre supérieur.

4.4.1 Le module de logique

Nous rappelons que nous avons utilisé le langage de programmation Objective Caml pour élaborer le démonstrateur. Ce langage donne la possibilité de gérer des modules, et de donner le type des éléments des modules. Nous donnons alors ici certains des éléments nécessaires au démonstrateur et qui sont fournis par le module de logique. Nous ne donnerons pas tout par souci de simplification.
Le fait qu'un type défini dans la logique soit abstrait signifie que tout module utilisant la logique n'a pas le moyen de connaître la structure des éléments de ce type ni de les gérer à sa guise. Il doit passer par les fonctions définies dans la logique pour le faire.

- **type formula** : le type des formules. Ce type est abstrait pour le démonstrateur ;
- **type substitution** : le type des substitutions, abstrait également pour le démonstrateur ;
- **apply_subst** : la fonction qui applique une substitution à une formule ;
- **exception Unif_fails** : dans le cas où une unification échoue, le module de logique doit lever une exception qui sera capturée par le démonstrateur ;
- **type constraints** : le type des contraintes. Celui-ci s'explique par le fait que l'on peut parfois souhaiter unifier seulement dans le cas où certaines contraintes sont vérifiées. Chaque clause contient alors ses propres contraintes, qui peuvent être comme nous le verrons plus tard un peu tout ce que l'on veut ;
- **type unif_kind** : comme le démonstrateur demande à unifier des formules pour des raisons autres que la règle de résolution, notamment pour la contraction et la subsomption, il est nécessaire que lors de l'unification on sache dans quel cas on se place. Le type d'unification est donc un type imposé par le démonstrateur ;
- **unif** : la fonction d'unification. Elle prend bien sûr deux formules en argument, connaît le type d'unification et les contraintes associées aux formules. Elle renvoie plusieurs choses :
  - la substitution (ou unificateur),
  - un entier qui est la taille de l'unificateur,
  - la formule unifiée,
  - une liste de formules, qui peuvent être des conditions à remplir pour avoir l'unification ;
- **size** : fonction qui donne la taille d'une formule, utilisée pour le calcul des poids des clauses ;
- **negative_formula** : fonction qui teste si une formule est négative à double négation près, c'est à dire que $\neg\neg A$ est positive si A est positive ;

- **negate_formula** : fonction qui rend la négation d'une formule ;
- **elim_all_neg** : fonction qui élimine toutes les négations d'une formule. On a besoin de ces trois dernières fonctions car les clauses sont partagées en deux ensembles, les formules positives et les formules négatives, et le démonstrateur ne traite que des formules sans négation. Comme il est inutile de conserver une double négation en logique classique (la résolution fait des preuves classiques), il est utile d'éliminer toutes les négations d'une formule ;
- **type renaming** : type pour le renommage de variables libres. En effet lorsque l'on cherche à appliquer la résolution entre deux clauses, les variables libres des deux clauses doivent être rendues disjointes comme nous l'avons vu plus haut ;
- **get_renaming_formula** et **rename_formula** : fonctions qui permettent de donner de nouveaux noms aux variables libres d'une formule ;
- **get_renaming_constraints** et **rename_constraints** : la même chose pour les contraintes si elles contiennent des formules ;
- **type varset** : le type d'un ensemble de variables ;
- **empty_varset** : l'ensemble de variables vide ;
- **vars_of_formula** : fonction qui donne les variables libres d'une formule ;
- **union_varset** : fait l'union d'ensembles de variables ;
- **vars_to_constants** : fonction qui renvoie une substitution transformant les variables en constantes. Cette fonction est utilisée pour tester la subsomption, puisque la formule subsumée ne doit pas être atteinte par une substitution ;
- **is_empty_inter_varset** : teste si deux ensembles de variables ont une intersection vide. Cette fonction est utilisée pour faire la séparation des clauses ;
- **get_rules** : donne la liste des règles de décomposition d'une formule associées au connecteur principal de celle-ci. Autrement dit donne toutes les règles applicables sur une formule, connaissant le fait qu'elle est positive ou négative. La fonction renvoie une liste d'éléments contenant
  - un entier qui est le poids de la règle,
  - une chaîne de caractère qui est le nom de la règle,
  - une liste de formules qui sont obtenues après l'utilisation de la règle,
  - une substitution pour le cas où la règle en met une en jeu,
  - une contrainte qui peut venir de la contrainte sur la formule donnée et de la règle ;
- **print_formula** : affiche une formule ;
- **print_constraints** : affiche une contrainte ;
- **equal_formula** : teste l'égalité de deux formules ;
- **head_symbol** : donne le symbole de tête d'une formule, utilisé pour améliorer la recherche de candidats pouvant s'unifier, les formules étant triées par symbole de tête.

4.4.2 Logique propositionnelle

C'est le module évidemment le plus facile. Les formules sont données par la grammaire

$$F ::= F \to F \,|\, F \leftrightarrow F \,|\, F \vee F \,|\, F \wedge F \,|\, \neg F \,|\, Atom$$

Où *Atom* est une variable propositionnelle. On peut également ajouter les formules "Vrai" et "Faux". Il n'y a pas de substitution, pas de contrainte et l'unification est le test de l'égalité. La taille d'une formule est le nombre de symboles (en dehors des parenthèses) de la formule. Il n'y a pas de renommage, pas d'ensemble de variables. Les règles sont les suivantes (pas de formule $\neg F$, et les formules atomiques n'ont pas de décomposition) :

1. Si la formule est positive
   - $F_1 \to F_2$: la clause $\neg F_1, F_2$
   - $F_1 \leftrightarrow F_2$: les clauses $F_1 \to F_2$ et $F_2 \to F_1$
   - $F_1 \vee F_2$: la clause F_1, F_2
   - $F_1 \wedge F_2$: les clauses F_1 et F_2.

2. Si la formule est négative
   - $F_1 \to F_2$: les clauses F_1 et $\neg F_2$
   - $F_1 \leftrightarrow F_2$: la clause $\neg(F_1 \to F_2), \neg(F_2 \to F_1)$
   - $F_1 \vee F_2$: les clauses $\neg F_1$ et $\neg F_2$
   - $F_1 \wedge F_2$: la clause $\neg F_1, \neg F_2$.

Le poids pour toutes les règles est le même. On remarque que pour l'équivalence nous avons appliqué uniquement les règles du $\wedge$. Nous aurions pu donner les clauses $\neg F_2, \neg F_1$ et F_1, F_2 pour le cas négatif. En effet, si l'on applique la règle négative pour l'implication sur $\neg(F_1 \to F_2), \neg(F_2 \to F_1)$ nous obtenons F_1, F_2, $F_1, \neg F_1$, $\neg F_2, F_2$ et $\neg F_2, \neg F_1$. Les tautologies pouvant être éliminées dans notre cas, il ne reste plus que les deux clauses $\neg F_2, \neg F_1$ et F_1, F_2. Or si l'on se rappelle qu'un littéral négatif signifie une formule à prouver, une équivalence négative signifie une équivalence à prouver. Pour prouver une équivalence en général on prouve d'abord une implication puis l'autre. Cette vision là est complètement perdue si l'on décompose trop. Par contre, avec l'aide de la séparation sans séparation, la décomposition de $F_1 \leftrightarrow F_2$ négative en $\neg(F_1 \to F_2), \neg(F_2 \to F_1)$ permet de séparer dans la plupart des cas cette clause en deux (c'est à dire lorsqu'il n'y a pas de variable libre, ce qui est évidemment toujours le cas en logique propositionnelle), ce qui permet de faire les deux preuves de manière indépendante.

4.4.3 Logique du premier ordre

Les formules sont données par la grammaire usuelle redéfinie dans l'annexe I, plus la relation d'équivalence $\leftrightarrow$. Les règles sont les mêmes que le cas propositionnel pour la partie propositionnelle. Pour le cas des quantificateurs, voici les règles :

1. Si la formule est positive
   - $\forall x.F(x)$: la clause $F(x)$ où x est une méta-variable, c'est à dire susceptible d'être substituée par un terme.
   - $\exists x.F(x)$: la clause $F(f(x_1, \ldots, x_n))$ où f est un nouveau symbole de fonction, et les x_i sont les (méta-)variables libres de $F(x)$ en dehors de x. Cela revient à mettre la formule en forme de Skolem.
2. Si la formule est négative
   - $\exists x.F(x)$: la clause $\neg F(x)$ où x est une méta-variable, c'est à dire susceptible d'être substituée par un terme.
   - $\forall x.F(x)$: la clause $\neg F(f(x_1, \ldots, x_n))$ où f est un nouveau symbole de fonction, et les x_i sont les (méta-)variables libres de $F(x)$ en dehors de x. Cela revient à mettre la formule en forme de Skolem.

Les justifications de ces règles seront données dans le chapitre 5. Nous avons considéré que l'on privilégiait davantage le raisonnement propositionnel, ce qui fait que les règles sur les quantificateurs ont un poids plus important. Ce sont elles qui peuvent rendre éventuellement la preuve très longue à trouver, car le nombre de clauses que l'on peut obtenir au premier ordre peut être infini, alors qu'il est toujours fini dans le cas propositionnel.

4.4.4 Le premier ordre et le langage restreint

Ayant voulu tester le langage restreint tel que défini dans le chapitre 2, nous avons implémenté ce langage dans la logique du premier ordre telle qu'elle est dans la section précédente. Certaines idées et problèmes ont émergé alors :

- Une des difficultés est le traitement des méta-variables introduites par les commandes. Il est considéré à chaque étape de démonstration que ces variables sont vues comme des constantes lorsque le démonstrateur doit valider les règles.
  Nous aimerions cependant dans certains cas que la machine trouve d'elle même une valeur aux méta-variables, du fait qu'il peut être fastidieux de donner à la main le bon terme.

C'est par exemple le cas lorsque l'on fait la preuve d'une formule du type $\forall\epsilon\exists\eta P(\epsilon,\eta)$, et que l'on cherche un η dépendant de ϵ vérifiant $P(\epsilon,\eta)$. Il peut arriver que le η qui convient dépende de manière un peu trop complexe, pour que l'on puisse le donner sans faire d'erreur, de ϵ. Ainsi à la fin de la preuve, l'utilisateur sait qu'il y a un η qui convient, mais ne peut pas dire précisément quelle est sa valeur.
Si cela n'a pas été implémenté à l'heure actuelle il y a cependant une piste, qui est de rajouter des contraintes à chaque clause indiquant la valeur actuelle des méta-variables.
Ainsi on contraint l'usage des résolutions au cas où les valeurs données aux méta-variables de chaque clause peuvent s'unifier. Lorsque l'on arrive à la clause vide la valeur finale est alors récupérée, et on met à jour l'ensemble des buts en substituant les méta-variables par les valeurs trouvées.
Bien sûr il est possible que la valeur trouvée ne soit pas celle que l'on souhaite. C'est aussi pour cela (en plus du manque de temps et d'autres contraintes liées à l'implémentation actuelle) que pour l'instant est laissé à l'utilisateur le soin de donner la valeur des méta-variables. Mais dans certains cas très simples, cela pourrait fonctionner comme on le souhaite.

- Il a fallu également gérer l'utilisation, indiquée par les commandes **by** et **with** du langage restreint, d'hypothèses par d'autres. En fait pour l'instant le démonstrateur ne gère pas cela bien que ce soit en projet, donc il a été décidé de faire cela dans la logique, en insérant à nouveau des contraintes permettant de gérer les utilisations de formules par d'autres, ou les valeurs de variables données pour les quantifications universelles.
  Ce qui a été fait est que chaque formule est associée à un nom, et si une formule F doit être utilisée par une autre K, le nom de F est donné dans les contraintes de K. Le nom des formules est gardé dans les décompositions successives dans les clauses. Lors de la résolution entre deux clauses pour l'une comportant dans ses contraintes le nom de F et pour l'autre ayant le nom de F, on considère alors que le poids de l'unificateur est très faible car on est en train d'utiliser une formule avec F, ce qui est indiqué par la contrainte.
  Dans le cas de l'utilisation d'une hypothèse avec un **with** $x = f(t)$, on met dans la contrainte l'égalité $x = f(t)$, et si l'on trouve une décomposition de l'hypothèse de la forme $\forall x.F(x)$, alors on décompose en donnant à x la valeur $f(t)$ en donnant à cette décomposition un poids très faible, et on fait également la décomposition habituelle, mais en donnant à celle-ci un poids très important.
- Le vieillissement des hypothèses a aussi été traité. Cela signifie que les hypothèses les plus anciennes ont un poids plus important que les plus récentes. Cela est d'autant plus vrai pour les hypothèses qui ont été nommées, puisque celles ci ne sont *a priori* utilisées que lorsqu'on le précise. Bien sûr ceci reste une hypothèse de travail, qui peut s'avérer mauvaise dans certains cas. Cependant elle peut se vérifier dans certains cas comme dans des preuves de livres de cours par exemple.

Toujours est-il qu'avec cette logique, nous bénéficions d'un assistant de démonstration de formules du premier ordre. Il faut noter cependant que nous n'avons pas traité le cas de l'égalité, qui rend les choses beaucoup plus complexes, notamment au niveau de l'unification. Le lecteur pourra voir dans l'annexe IV une preuve écrite dans cette logique.

4.4.5 PhoX et la logique d'ordre supérieur

Comme nous l'avons signalé dans le chapitre 1, le but du démonstrateur était d'améliorer la preuve automatique de PhoX dans le cas des new_command, qui sont une première version du langage restreint. Récemment notre démonstrateur a donc été intégré à l'assisant de démonstration, et peut optionnellement remplacer la tactique trivial d'origine. Ainsi le démonstrateur peut être utilisé pour prouver des formules, ou pour justifier les new_ command. Il faut cependant signaler que rien n'a été fait sur les new_command. C'est à dire que le langage n'est pas celui du langage restreint, et que l'interprétation n'est pas celle que nous avons donné dans le chapitre 2. En particulier les commandes utilisant **by** et **with** ne sont

pas du tout interprétées. Ceci est dû au fait que le démonstrateur n'a pas intégré cela, comme nous l'avons expliqué dans la section précédente.
La logique de PhoX est une logique d'ordre supérieur avec égalité, donc les difficultés pour le démonstrateur sont plus grandes encore que pour le premier ordre. Aux premiers essais il s'est avéré que la rapidité des deux démonstrateurs est inégale. Dans certains cas la tactique trivial est plus rapide, et dans d'autre cas elle est plus lente que le démonstrateur.
Ce qui est plutôt encourageant est le fait que le démonstrateur est jeune, et donc n'est pas encore optimisé. De plus, certaines modifications sur le calcul du poids des clauses montre que le démonstrateur peut gagner beaucoup de temps sur certaines preuves. Il faut donc parvenir à optimiser le calcul des poids pour pouvoir espérer une meilleure efficacité.
Une telle optimisation pourrait être faite en utilisant des algorithmes génétiques, qui permettent de trouver de bons paramètres. Il suffirait de donner une base de formules à prouver, et de faire modifier des paramètres entrant en jeu dans le calcul des poids afin que le temps de résolution des formules soit le plus court possible.
Une autre chose qui permet d'espérer que le démonstrateur devienne efficace est que l'intégration du démonstrateur dans PhoX s'est faite rapidement. Par exemple, certains paramètres demandés comme le poids des règles et l'indice (un paramètre indiquant si la règle peut être utilisée souvent ou rarement, voir l'annexe III) sont des valeurs constantes, alors que l'on peut sans doute améliorer cela, certaines règles pouvant être considérées comme rarement utilisées ou couteuses.

Chapitre 5

Un système logique

5.1 Ce que l'on souhaite

Comme nous l'avons vu, la méthode utilisée par le démonstrateur n'est pas standard, dans le sens où c'est un démonstrateur par résolution qui décompose les formules en cours de preuve et non au préalable. C'est à dire qu'il n'utilise pas seulement les deux seules règles utilisées pour la résolution, à savoir la règle de résolution et la règle de contraction, mais également des règles de décomposition.
Le but de ce chapitre est de donner un cadre théorique à l'implémentation qui a été faite. Nous allons donner les règles de deux systèmes de démonstration qui proviennent de cette observation :

- Le premier, que nous appellerons système logique, est un système dual du calcul des séquent, dans le sens où il ne contient que des règles d'élimination quand le calcul des séquents ne contient que des règles d'introduction. En effet ce sont les règles d'élimination qui décomposent les formules.
  Cette dualité implique que le système logique n'a pas de règle axiome, alors que dans le calcul des séquents la règle de coupure peut être éliminée. La règle de coupure a autant d'importance dans le système logique que la règle axiome dans le calcul des séquents : il s'agit en fait de la règle de résolution sur laquelle repose le système.
- Le second système, que nous appelerons système implémenté, lui ressemble dans sa présentation, mais il est plus proche de ce qui a été effectivement implémenté. Il s'éloigne d'un système de déduction logique par le fait que les règles peuvent faire intervenir des substitutions.

Pour chacun des systèmes
– nous prouvons la complétude, ce qui dans notre cas revient à montrer leur équivalence avec le calcul des séquents classique ;
– nous donnerons une stratégie complète.

Pour le système logique les preuves sont syntaxiques, alors que pour le système implémenté elles sont sémantiques.

5.2 Le système logique

Nous donnons ici le système de démonstration pour la logique du premier ordre.
Il y a dans le système des symboles primitifs que nous devons expliquer.
– Nous appellerons littéral une formule.
– le symbole $\perp$ mis en exposant à un littéral F dénote le fait que F est négatif.
– Un (multi)-ensemble de littéraux, noté A_1,A_2,...,A_n ou Γ ou encore Δ s'appelle une clause. Une clause représente la disjonction des formules qui la compose.
– Un ensemble de clauses est noté Γ_1 ;Γ_2 ;... ;Γ_3 ou bien S. Un ensemble de clauses représente la conjonction des clauses qui le compose.

– La clause vide sera notée $\square$.

Toutes les règles suivantes sont des règles de déduction d'un ensemble de clauses vers un ensemble de clauses. A chaque règle on ajoute une clause, et aucune n'est éliminée.
Pour simplifier l'apparence des règles nous ne notons que la (les) clause(s) principale(s) (que nous appellerons également clause(s) active(s)) de l'ensemble de clauses qui subit la règle de déduction en prémisse, et la clause obtenue en conclusion. Par exemple pour la règle $\vee_p$, si l'on a l'ensemble de clauses $S = S'; \Gamma, A \vee B$, alors nous obtenons l'ensemble de clauses $S; \Gamma, A, B$.

$$\frac{\Gamma, A \vee B}{\Gamma, A, B} \vee_p \quad \frac{\Gamma, (A \vee B)^\perp}{\Gamma, A^\perp} \vee_{n_g} \quad \frac{\Gamma, (A \vee B)^\perp}{\Gamma, B^\perp} \vee_{n_d}$$

$$\frac{\Gamma, A \to B}{\Gamma, A^\perp, B} \to_p \quad \frac{\Gamma, (A \to B)^\perp}{\Gamma, A} \to_{n_g} \quad \frac{\Gamma, (A \to B)^\perp}{\Gamma, B^\perp} \to_{n_d}$$

$$\frac{\Gamma, (A \wedge B)^\perp}{\Gamma, A^\perp, B^\perp} \wedge_n \quad \frac{\Gamma, A \wedge B}{\Gamma, A} \wedge_{p_g} \quad \frac{\Gamma, A \wedge B}{\Gamma, B} \wedge_{p_d}$$

$$\frac{\Gamma, (\neg A)^\perp}{\Gamma, A} \neg_n \qquad \frac{\Gamma, \neg A}{\Gamma, A^\perp} \neg_p$$

$$\frac{\Gamma, \forall x.A(x)}{\Gamma, A(t)} \forall_p \qquad \frac{\Gamma, (\forall x.A(x))^\perp}{\Gamma, A(y)^\perp} \forall_n(*)$$

$$\frac{\Gamma, \exists x.A(x)}{\Gamma, A(y)} \exists_p(*) \qquad \frac{\Gamma, (\exists x.A(x))^\perp}{\Gamma, A(t)^\perp} \exists_n$$

$$\frac{\Gamma, A, A}{\Gamma, A} \mathit{contr}_p \qquad \frac{\Gamma, A^\perp, A^\perp}{\Gamma, A^\perp} \mathit{contr}_n$$

$$\frac{\Gamma, A; \Gamma', A^\perp}{\Gamma, \Gamma'} \mathit{res}$$

(*) y est une variable fraîche, c'est à dire y n'est pas libre dans S, Γ et $A(x)$

Remarque 5.2.1 Deux remarques sur ce système :

1. à l'image de la déduction libre (voir [Pa]) et comme nous l'avons déjà signalé, les règles sont des règles d'élimination ;
2. à l'image du calcul des structures (voir [BruTi]), il n'y a pas de branchement des règles. D'autre part la structure globale de l'ensemble de clauses est une conjonction de disjonctions. Il ne s'agit cependant pas de faire de l'inférence profonde, bien au contraire.

Note 5.2.2 Les règles contr_p et contr_n seront souvent par la suite confondues, nous noterons alors *contr* l'une quelconque de ces deux règles.

5.3 Quelques définitions et premières propriétés

Définition 5.3.1 *Dire que l'on peut dériver une clause Γ à partir d'un ensemble de clauses S signifie qu'il existe un ensemble de clauses T et une liste d'ensembles de clauses tels que :*
– *le premier élément de la liste est l'ensemble S ;*
– *Si deux ensembles sont successifs dans la liste il existe une règle de déduction qui permet de passer du premier au suivant ;*

– le dernier ensemble de la liste est l'ensemble $S; T; \Gamma$.
Cette liste est appelée une dérivation.

Définition 5.3.2 *Les règles autres que res et contr sont dites règles de décomposition.*

Nous n'écrirons pas les dérivations comme des listes, mais comme des arbres à l'image des règles de déduction. Ceux-ci étant linéaires, nous pouvons considérer qu'il y a une règle la plus haute et une règle la plus basse.

Définition 5.3.3 *Dans toute dérivation dans notre système, nous appelons*
– première règle de la dérivation la règle la plus haute dans l'écriture de la dérivation ;
– dernière règle la règle la plus basse dans l'écriture de la dérivation.

Lemme 5.3.4 *Soit S un ensemble de clauses, T un autre ensemble de clauses et Γ une clause. Supposons qu'il existe une dérivation de Γ à partir de S. Alors il existe une dérivation de Γ à partir de $S; T$.*

Preuve : les règles ne faisant aucune hypothèse sur l'ensemble de clauses contenant la (les) clause(s) active(s), il est tout à fait possible d'y ajouter toutes les clauses que l'on souhaite. □

Définition 5.3.5 (arbre de dérivation) *La dérivation d'une clause Δ à partir d'un ensemble de clauses S permet de définir un arbre dont Δ est la racine, dont les noeuds sont des clauses et dont les branches sont étiquetées par les règles.*
On appelle cet arbre l'arbre de dérivation associé à Δ. Les clauses qui sont aux noeuds et aux feuilles de l'arbre sont appelées les ancêtres de Δ. Cet arbre est défini par induction sur la dérivation, en regardant la dernière règle.
– si Δ est dans S (pas de règle utilisée), alors l'arbre est composé de la seule racine Δ ;
– si la dernière règle n'introduit pas Δ alors il y a une dérivation plus courte de Δ, et l'induction termine ;
– si la dernière règle R introduit Δ alors
  – si ce n'est pas la résolution, Δ provient d'une seule clause, dont par induction on peut prendre l'arbre de dérivation T, alors l'arbre est

$$\dfrac{T}{\Delta}\, R$$

  – si c'est la résolution, Δ provient de deux clauses, dont par induction on peut prendre les arbres de dérivation T_1 et T_2, alors l'arbre est

$$\dfrac{T_1 \quad T_2}{\Delta}\, res$$

Remarque 5.3.6 Si à toute dérivation on peut associer un arbre, on ne peut pas en déduire un système de dérivation par arbre sans donner plus de conditions. Car si nous prouvons qu'une dérivation est sûre dans la sous-section 5.4.2, les preuves sous forme d'arbre ne le sont pas. En effet, on peut voir que si l'on crée un arbre à partir des règles, on peut aboutir à ceci :

$$\dfrac{\dfrac{\dfrac{((\exists x.A(x)) \to (\forall x.A(x)))^{\perp}}{\exists x.A(x)} \to_{n_g}}{A(y)}\, \exists_p \qquad \dfrac{\dfrac{((\exists x.A(x)) \to (\forall x.A(x)))^{\perp}}{(\forall x.A(x))^{\perp}} \to_{n_d}}{A(y)^{\perp}}\, \forall_n}{\square}\, res$$

Les règles respectent la condition de 'variable fraîche', mais nous ont permis d'écrire un arbre de déduction ayant pour racine la clause vide alors que

$$\nvdash_{LK} (\exists x.A(x)) \to (\forall x.A(x))$$

Dans le système logique que nous avons décrit nous serions dans ce cas obligé d'avoir deux variables distinctes car les règles $\exists_p$ et $\forall_n$ sont utilisées l'une après l'autre. Une solution éventuelle serait de donner une variable différente pour chaque formule décomposée, ce qui interdirait la réutilisation d'une même variable.
Nous ne donnerons donc pas de système de démonstration sous forme d'arbre. D'autre part, notre volonté d'être proche de ce qui a été implémenté conduit naturellement au système décrit. La vision d'une preuve par arbre n'est alors intéressante que pour un affichage compréhensible des dérivations et pour l'usage théorique que nous allons en faire.

Définition 5.3.7 (utilisation d'une clause dans une dérivation) *Soit S un ensemble de clauses. Supposons qu'il existe une dérivation d'une clause Δ à partir de S. Soit Γ une clause obtenue lors de cette dérivation. On dit que cette dérivation utilise Γ si Γ est un ancêtre de Δ dans l'arbre de dérivation.*

Définition 5.3.8 (Dérivation utile) *Nous appellerons dérivation utile la dérivation d'une clause Δ à partir d'un ensemble S si toutes les clauses de S sont des ancêtres de Δ et si toutes les clauses obtenues lors de la dérivation sont des ancêtres de Δ.*

Lemme 5.3.9 *Soit Δ une clause et S un ensemble de clauses. Supposons qu'il existe une dérivation de la clause Δ à partir de S. Alors il existe un ensemble S' inclus dans S et une dérivation utile de Δ à partir de S'. De plus, cette dérivation est plus courte (ou égale) à la dérivation initiale.*

Preuve : Nous pouvons supposer que Δ n'appartient pas à S, sinon il suffit de prendre $S' = \Delta$ et c'est fini. Nous faisons alors la preuve par induction sur la dérivation, en regardant la première règle R.
Comme Δ n'est pas dans S il y a au moins une règle appliquée, ce qui nous donne un ensemble de clauses $S; \Gamma$, et une dérivation de la clause Δ plus courte. Nous avons alors par induction une dérivation de Δ à partir d'un ensemble d'ancêtres S'' de Δ inclus dans $S; \Gamma$ qui est utile. Deux cas

- Si S'' contient Γ, alors Γ est un ancêtre de Δ et S'' s'écrit $S'_1; \Gamma$ avec S'_1 inclus dans S. La clause (ou les clauses dans le cas de la résolution) sur laquelle a été appliquée la règle R est un ancêtre de Δ dans ce cas et nous la rajoutons à S'_1 si nécessaire pour donner un ensemble S' d'ancêtres de Δ inclus dans S. On a alors

$$\dfrac{\dfrac{\dfrac{S'}{S'; \Gamma}\, R}{\vdots}}{\Delta}$$

  où toutes les clauses sont des ancêtres de Δ.
- Si S'' ne contient pas Γ, alors S'' est inclus dans S et nous avons le résultat. □

Lemme 5.3.10 *Soit S' un ensemble de clauses, Δ une clause et F une formule. Supposons qu'il existe une dérivation de la clause vide à partir de $S = S'; \Delta$ utilisant Δ, alors il existe une dérivation de la clause F à partir de $S'; \Delta, F$*

Preuve : Par induction sur la taille de la dérivation, en regardant la première règle. Nous pouvons supposer quitte à renommer les variables introduites par les règles $\forall_n$ ou $\exists_p$, que celles-ci sont distinctes des variables libres de F. Si la première règle de la dérivation ne porte pas sur Δ, alors l'induction termine. Sinon, Δ s'écrit Γ, G avec G une formule. Pour toutes les règles on peut ajouter F à Γ, la règle reste valide en ajoutant F à la clause obtenue, y compris pour $\forall_n$ et $\exists_p$ grâce à ce que nous avons supposé plus haut. Par exemple

$$\frac{S = S'; \Gamma, A \vee B}{S; \Gamma, A, B} \vee_p \quad \text{devient} \quad \frac{S = S'; \Gamma, F, A \vee B}{S; \Gamma, F, A, B} \vee_p$$

$$\frac{S = S'; \Gamma, A; \Gamma', A^\perp}{S; \Gamma, \Gamma'} res \quad \text{devient} \quad \frac{S'; \Gamma, F, A; \Gamma', A^\perp}{S; \Gamma, F, \Gamma'} res$$

On peut supposer d'après le lemme 5.3.9 que la clause obtenue est utilisée dans la dérivation. L'induction permet alors de terminer. □

On peut remarquer que les règles $\exists_p$ ou $\forall_n$ peuvent être *a priori* utilisées plusieurs fois sur la même formule d'une même clause pour aboutir à la clause vide. Nous allons montrer après ce lemme qu'il est en fait inutile de le faire.

Lemme 5.3.11 *Soit F une formule contenue dans une clause $C = F, C'$ (ou $F^\perp, C'$). Soit σ une substitution.*

1. *Supposons qu'une règle R (différente de res et de contr) est applicable sur la formule F de C. Alors cette même règle est applicable sur la clause $C\sigma$.*
2. *Soit $D = F^\perp, D'$ (ou F, D') une autre clause. La règle res applicable sur la formule F dans les clauses C et D est également applicable sur la formule $F\sigma$ dans les clauses $C\sigma$ et $D\sigma$.*
3. *Si $C = F, F, C''$ ou $F^\perp, F^\perp, C''$ alors la règle contr applicable dans la clause C est applicable sur la clause $C\sigma$.*

Dans les trois cas, si avant substitution la clause obtenue est Γ, alors après substitution la clause obtenue est $\Gamma\sigma$.

Preuve : Les deuxième et troisième points sont évidents. Le premier point est prouvé en regardant la tête de la formule F. Comme une règle est applicable, F n'est pas une formule atomique (pas de règle de décomposition pour les formules atomiques).
Si $F = F_1 \mathcal{R} F_2$ avec $\mathcal{R}$ un symbole de relation binaire, alors $F\sigma = F_1\sigma \mathcal{R} F_2\sigma$, donc la règle peut être appliquée. Si $F = \neg F_1$ alors $F\sigma = \neg F_1\sigma$ et la règle peut être appliquée. Enfin si $F = ?x.F_1(x)$ avec ? un quantificateur, alors $F\sigma = ?z.F_1(z)\sigma$ avec z une variable fraîche (pour éviter les captures). Alors la règle est applicable sur $F\sigma$.
On obtient bien $\Gamma\sigma$ dans tous les cas. □

Lemme 5.3.12 *Soit S un ensemble de clauses. Supposons qu'il existe une dérivation de l'ensemble de clauses S' à partir de S. Alors il existe une substitution σ et une déduction de l'ensemble de clauses $S'\sigma$ à partir de S telle qu'aucune règle $\exists_p$ ou $\forall_n$ n'est appliquée plus d'une fois sur une même formule d'une même clause. De plus cette preuve est plus courte. En particulier, si S' contient la clause vide alors il existe une dérivation de la clause vide avec cette restriction.*

Preuve : Par induction sur la taille de la preuve, en regardant la dernière règle R. Si aucune règle n'est utilisée alors σ est l'identité. Sinon une règle R a été appliquée pour donner la clause Γ ajoutée à un ensemble de clauses S'' tel que $S' = \Gamma; S''$. Par induction il existe une substitution σ' et une déduction de $S''\sigma'$ à partir de S vérifiant la restriction. D'après le lemme précédent, la règle R peut encore être applicable, pour obtenir $\Gamma\sigma'$.
Si la règle R est différente de $\exists_p$ ou $\forall_n$ alors on peut l'appliquer et les conditions sont bien remplies. Dans le cas contraire
- si la clause sur laquelle la règle est appliquée n'a jamais subit la même règle sur la même formule, nous pouvons appliquer également la règle et conserver les conditions.
- Sinon la règle a déjà été appliquée. Par hypothèse elle ne l'a été qu'une fois, en introduisant une clause $F(x), C$ (ou $F(x)^\perp, C$) avec x une variable fraîche, qui se trouve dans $S''\sigma'$. Appliquer R introduit alors la clause $\Gamma\sigma' = F(y), C$ (ou $F(y)^\perp, C$) avec y une variable fraîche.

Si l'on définit σ comme étant σ' sur toutes les variables sauf y et $\sigma(y) = x$ alors $S''\sigma = (\Gamma; S'')\sigma$ car $\Gamma\sigma = F(x), C$ (ou $F(x)^\perp, C$). Or $\Gamma; S'' = S'$ donc il y a une déduction de $S'\sigma$ à partir de S vérifiant les conditions car sans utiliser la dernière règle R. □

Ainsi, toutes les règles peuvent n'être utilisées qu'une fois (on considère que chaque utilisation des règles $\exists_n$ et $\forall_p$ avec un terme différent est une règle différente). En effet chaque autre règle appliquée à une clause donnée forme une clause déterminée, n'ajoutant aucun élément nouveau. Comme l'ensemble de clauses est un ensemble, répéter deux fois une même règle revient à faire l'identité.

5.4 Complétude du système logique

Nous souhaitons prouver que ce système est sûr et complet. Expliquons le sens de ces mots : Ce que nous appelons démonstration dans ce système est la dérivation de la clause vide à partir d'un ensemble de clauses. Il faut bien se rappeler que le principe de la résolution repose essentiellement sur le tiers exclu et donc sur la logique classique. On cherche une contradiction dans un ensemble de clauses, et donc pour prouver F, on suppose que l'on a $\neg F$ et on cherche une contradiction. Ici donc, si l'on cherche à prouver la formule F, il faut dériver la clause vide à partir de la seule clause $F^\perp$.
Dire que le système est sûr, c'est dire que si l'on a dérivé la clause vide à partir de $F^\perp$, alors F est valide. Dire que le système est complet, c'est dire que si F est valide alors il existe une dérivation de la clause vide à partir de $F^\perp$.
Pour prouver cela on peut se ramener à un système dont on sait qu'il est sûr et complet. Comme la résolution repose sur la logique classique, nous devons choisir un système classique. Comme nous supposons que notre ensemble de clauses est contradictoire, cela signifie, si l'on considère que l'ensemble de clauses est interprété comme une formule sous forme normale conjonctive (conjonction de disjonctions de formules), que la négation de cette formule est valide. Autrement dit, en entrant la négation à l'intérieur, la disjonction des conjonctions des négations des formules est valide.
Il y a un système qui permet d'écrire de manière intéressante la disjonction de formules. Il s'agit du calcul des séquents, puisque l'on peut avoir plusieurs conclusions dans un séquent, ce qui représente la disjonction de celles-ci.
Nous chercherons donc à prouver l'équivalence de notre système avec le calcul des séquents classique LK. Pour fixer les notations, rappelons ici les règles de ce système :

$$\frac{}{A \vdash A}\, ax \qquad \frac{\Gamma \vdash A, \Delta \quad \Gamma', A \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\, cut$$

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \wedge_{g_d} \quad \frac{\Gamma, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \wedge_{g_g} \quad \frac{\Gamma \vdash A, \Delta \quad \Gamma' \vdash B, \Delta'}{\Gamma, \Gamma' \vdash A \wedge B, \Delta, \Delta'} \wedge_d$$

$$\frac{\Gamma, A \vdash \Delta \quad \Gamma', B \vdash \Delta'}{\Gamma, \Gamma', A \vee B \vdash \Delta, \Delta'} \vee_g \quad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash A \vee B, \Delta} \vee_{d_g} \quad \frac{\Gamma \vdash B, \Delta}{\Gamma \vdash A \vee B, \Delta} \vee_{d_d}$$

$$\frac{\Gamma, B \vdash \Delta \quad \Gamma' \vdash A, \Delta'}{\Gamma, \Gamma', A \to B \vdash \Delta, \Delta'} \to_g \quad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash A \to B, \Delta} \to_{d_g} \quad \frac{\Gamma \vdash B, \Delta}{\Gamma \vdash A \to B, \Delta} \to_{d_d}$$

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta} \neg_g \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta} \neg_d$$

$$\frac{\Gamma, A[t/x] \vdash \Delta}{\Gamma, \forall x A \vdash \Delta} \forall_g \qquad \frac{\Gamma \vdash A[y/x], \Delta}{\Gamma \vdash \forall x A, \Delta} \forall_d(*)$$

$$\frac{\Gamma, A[y/x] \vdash \Delta}{\Gamma, \exists x A \vdash \Delta}\,\exists_g(*) \qquad \frac{\Gamma \vdash A[t/x], \Delta}{\Gamma \vdash \exists x A, \Delta}\,\exists_d$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}\,w_g \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash A, \Delta}\,w_d$$

$$\frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta}\,c_g \qquad \frac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta}\,c_d$$

(*) y n'est pas libre dans Γ, A, Δ

Remarque 5.4.1 Il est à noter déjà que la règle de coupure de LK est la même que la règle de résolution, quand on voit le séquent $A, B \vdash C, D$ comme la formule $\neg A \vee \neg B \vee C \vee D$, c'est à dire comme la clause $A^{\perp}, B^{\perp}, C, D$.

5.4.1 Notations

Par la suite nous adopterons des notations pour simplifier l'écriture des clauses. Une clause étant un ensemble de formules, nous pouvons regrouper les formules négatives ensemble et les formules positives ensemble. Nous aurons alors la notation globale $\Gamma^{\perp}, \Delta$ pour une clause, où $\Gamma^{\perp} = \{G_i^{\perp}\}_{i \in I}$ est l'ensemble des formules négatives et $\Delta = \{H_j\}_{j \in J}$ l'ensemble des formules positives.
Un ensemble de clauses sera noté $S = \{\Gamma_k^{\perp}, \Delta_k\}_{k=1..n}$.
Nous noterons $\Gamma \vdash_{LK} \Delta$ le fait que $\Gamma \vdash \Delta$ soit dérivable dans LK.
Lorsqu'une règle de dérivation R est utilisée potentiellement plusieurs fois pour passer d'une étape à une autre lors d'une dérivation, nous utiliserons la notation R^*.

Pour traduire notre système dans LK, nous avons besoin de définitions d'une fonction transformant les clauses en formules.

Définition 5.4.2 *Soit $\Gamma^{\perp}, \Delta$ une clause non vide. Alors on définit*

$$\psi(\Gamma^{\perp}, \Delta) = (\wedge_i G_i) \wedge (\wedge_j \neg H_j)$$

On remarque que ψ appliqué à une clause renvoie donc sa négation. Si S est un ensemble de clauses $\{\Gamma_k^{\perp}, \Delta_k\}_{k=1..n}$ nous notons $\psi(S)$ l'ensemble des images des clauses de S par ψ :

$$\psi(S) = \{\psi(\Gamma_k^{\perp}, \Delta_k)\}_{k=1..n}$$

Remarque 5.4.3 Nous utiliserons le plus souvent sans le dire le fait que les clauses sont des multi-ensembles, donc sont en particulier non ordonnées. Cela nous permet de les écrire dans l'ordre qui nous convient. Le fait que dans LK le $\wedge$ est commutatif et associatif nous assure que cela ne pose aucun problème pour la définition de ψ et tous les choix que l'on pourra faire seront valides.

5.4.2 Le système logique est sûr

Nous souhaitons prouver que si l'on déduit la clause vide à partir de $F^{\perp}$, alors $\vdash_{LK} F$. Pour cela, nous allons prouver un fait plus général.

Proposition 5.4.4 *Soit $S = \{\Gamma_k^{\perp}, \Delta_k\}_{k=1..n}$ un ensemble de clauses non vides. Supposons que l'on puisse dériver la clause vide à partir de S. Alors $\vdash_{LK} \psi(S)$.*

Preuve : Celle-ci se fait par induction sur la longueur de la dérivation de la clause vide à partir de S. Comme S ne contient pas la clause vide, il y a nécessairement une règle qui est utilisée. C'est à dire que la dérivation ressemble à

$$\dfrac{\dfrac{\dfrac{S}{S;C}\,R}{\vdots}}{\square}$$

Si C est la clause vide, cela signifie que R est la règle de résolution, et qu'il existe une formule F telle que F et $F^\perp$ soient des clauses de S. Alors S peut s'écrire $S = S'; F; F^\perp$. Nous voulons alors prouver que $\vdash_{LK} \psi(S'), \neg F, F$, ce qui est le cas :

$$\dfrac{\dfrac{\dfrac{}{F \vdash F}\,ax}{\vdash F, \neg F}\,\neg_d}{\vdash \psi(S'), F, \neg F}\,w^*_d$$

Si C n'est pas vide, alors $S; C$ est un ensemble de clauses non vides à partir duquel on peut dériver la clause vide avec une longueur de dérivation plus courte que pour S. Ainsi l'hypothèse d'induction nous permet de conclure que $\vdash_{LK} \psi(S), \psi(C)$. Nous devons prouver que de $\vdash \psi(S), \psi(C)$ on peut déduire $\vdash \psi(S)$. Nous noterons $\psi(\Gamma) = \Pi$ pour Γ une clause et $\psi(S') = \Sigma$ pour S' un ensemble de clauses. Voyons donc les différents cas selon la règle R. Nous donnons des preuves syntaxiques, mais simplifiées compte tenu de la propriété de commutativité et d'associativité du $\wedge$. Nous ne donnons que quelques cas, les autres se trouvant à l'annexe V :

– $R = \vee_p$: S contient une clause $\Gamma, A \vee B$ et $C = \Gamma, A, B$. Nous avons donc par hypothèse $\vdash_{LK} \Sigma, \Pi \wedge \neg(A \vee B), \Pi \wedge \neg A \wedge \neg B$. Définissons T comme

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,ax}{\neg A, A \vdash}\,\neg_g}{\Pi \wedge \neg A, A \vdash}\,\wedge_{g_g}}{\Pi \wedge \neg A \wedge \neg B, A \vdash}\,\wedge_{g_d} \quad \dfrac{\dfrac{\dfrac{}{B \vdash B}\,ax}{\neg B, B \vdash}\,\neg_g}{\neg A \wedge \neg B, B \vdash}\,\wedge_{g_g}}{\Pi \wedge \neg A \wedge \neg B, B \vdash}\,\wedge_{g_g}}{\Pi \wedge \neg A \wedge \neg B, \Pi \wedge \neg A \wedge \neg B, A \vee B \vdash}\,\vee_g}{\Pi \wedge \neg A \wedge \neg B, A \vee B \vdash}\,c_g}{\Pi \wedge \neg A \wedge \neg B \vdash \neg(A \vee B)}\,\neg_d \qquad \vdash \Sigma, \Pi \wedge \neg(A \vee B), \Pi \wedge \neg A \wedge \neg B}{\vdash \Sigma, \Pi \wedge \neg(A \vee B), \neg(A \vee B)}\,cut$$

Alors

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi \vdash \Pi}\,ax}{\Pi \wedge \neg A \vdash \Pi}\,\wedge_{g_d}}{\Pi \wedge \neg A \wedge \neg B \vdash \Pi}\,\wedge_{g_d} \qquad \vdash \Sigma, \Pi \wedge \neg(A \vee B), \Pi \wedge \neg A \wedge \neg B}{\vdash \Sigma, \Pi \wedge \neg(A \vee B), \Pi}\,cut \qquad T}{\vdash \Sigma, \Sigma, \Pi \wedge \neg(A \vee B), \Pi \wedge \neg(A \vee B), \Pi \wedge \neg(A \vee B)}\,\wedge_d}{\vdash \Sigma, \Pi \wedge \neg(A \vee B)}\,c^*_d$$

– $R = \forall_p$: S contient une clause $\Gamma, \forall x.A(x)$ et $C = \Gamma, A(t)$. Nous avons donc par hypothèse $\vdash_{LK} \Sigma, \Pi \wedge \neg\forall x.A(x), \Pi \wedge \neg A(t)$. Définissons T comme

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A(t) \vdash A(t)}\,ax}{\neg A(t), A(t) \vdash}\,\neg_g}{\Pi \wedge \neg A(t), A(t) \vdash}\,\wedge_{g_g} \qquad \vdash \Sigma, \Pi \wedge \neg\forall x.A(x), \Pi \wedge \neg A(t)}{A(t) \vdash \Sigma, \Pi \wedge \neg\forall x.A(x)}\,cut}{\dfrac{\forall x.A(x) \vdash \Sigma, \Pi \wedge \neg\forall x.A(x)}{\vdash \Sigma, \Pi \wedge \neg\forall x.A(x), \neg\forall x.A(x)}\,\neg_d}\,\forall_g$$

Alors

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi \vdash \Pi}\,ax}{\Pi \wedge \neg A(t) \vdash \Pi}\,\wedge_{g_d} \qquad \vdash \Sigma, \Pi \wedge \neg\forall x.A(x), \Pi \wedge \neg A(t)}{\vdash \Sigma, \Pi \wedge \neg\forall x.A(x), \Pi}\,cut \qquad T}{\vdash \Sigma, \Sigma, \Pi \wedge \neg\forall x.A(x), \Pi \wedge \neg\forall x.A(x), \Pi \wedge \neg\forall x.A(x)}\,\wedge_d}{\vdash \Sigma, \Pi \wedge \exists x.A(x)}\,c_d^*$$

– $R = \forall_n : S$ contient une clause $\Gamma, (\forall x.A(x))^\perp$ et $C = \Gamma, A(y)^\perp$ avec y une variable fraîche. Nous avons donc par hypothèse $\vdash_{LK} \Sigma, \Pi \wedge \forall x.A(x), \Pi \wedge A(y)$. Définissons T comme

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A(y) \vdash A(y)}\,ax}{\Pi \wedge A(y) \vdash A(y)}\,\wedge_{g_g} \qquad \vdash \Sigma, \Pi \wedge \forall x.A(x), \Pi \wedge A(y)}{\vdash \Sigma, \Pi \wedge \forall x.A(x), A(y)}\,cut}{\vdash \Sigma, \Pi \wedge \forall x.A(x), \forall y.A(y)}\,\forall_d(*)$$

Alors

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi \vdash \Pi}\,ax}{\Pi \wedge A(y) \vdash \Pi}\,\wedge_{g_d} \qquad \vdash \Sigma, \Pi \wedge \forall x.A(x), \Pi \wedge A(y)}{\vdash \Sigma, \Pi \wedge \forall x.A(x), \Pi}\,cut \qquad T}{\vdash \Sigma, \Sigma, \Pi \wedge \forall x.A(x), \Pi \wedge \forall x.A(x), \Pi \wedge \forall x.A(x)}\,\wedge_d}{\vdash \Sigma, \Pi \wedge \forall x.A(x)}\,c_d^*$$

(*) la condition est bien remplie.

– Voir l'annexe V pour les autres cas. □

5.4.3 Le système logique est complet

On veut prouver que si $\vdash_{LK} F$ alors il existe une dérivation de la clause vide à partir de $F^\perp$. Pour cela nous allons prouver un fait plus général.

Proposition 5.4.5 *Soit $\{H_i\}_{i\in\{1,\ldots,k\}}$ et $\{F_j\}_{j\in\{1,\ldots,l\}}$ des formules. Si l'on a*

$$H_1, \ldots, H_k \vdash_{LK} F_1, \ldots, F_l$$

alors il existe une dérivation de la clause vide à partir de l'ensemble de clauses

$$S = H_1; \ldots; H_k; F_1^\perp; \ldots; F_l^\perp$$

Preuve : Nous savons que le système LK vérifie l'élimination des coupures (voir plus tard la proposition 5.7.3). Nous faisons donc la preuve par induction sur la taille de la dérivation de $H_1, \ldots, H_k \vdash F_1, \ldots, F_l$ sans coupure. Raisonnons selon la dernière règle. Nous ne donnons pas tous les cas, le reste de la preuve étant reportée à l'annexe V :

– $\dfrac{\Gamma, A[t/x] \vdash \Delta}{\Gamma, \forall x A \vdash \Delta}\,\forall_g$: Par hypothèse d'induction, il existe une dérivation de la clause vide à partir de $T = S_1; A(t)$.
Or par la règle $\forall_p$ nous obtenons l'ensemble de clauses
$S_1; A(t); \forall x.A(x)$ à partir de $S_1; \forall x.A(x)$ et nous concluons grâce au lemme 5.3.4.
– $\dfrac{\Gamma \vdash A[y/x], \Delta}{\Gamma \vdash \forall x A, \Delta}\,\forall_d$: Par hypothèse d'induction, il existe une dérivation de la clause vide à partir de $T = S_1; A(y)^{\perp}$.
Or par la règle $\forall_n$ nous obtenons l'ensemble de clauses $S_1; A(y)^{\perp}; (\forall x.A(x))^{\perp}$ à partir de $S_1; (\forall x.A(x))^{\perp}$ et nous concluons grâce au lemme 5.3.4. Remarquons que la condition sur la règle $\forall_d$ implique que la condition sur la règle $\forall_n$ est respectée.
– Voir le reste des cas dans l'annexe V. □

5.5 Une stratégie complète du système logique

5.5.1 L'élimination des clauses subsumées

Nous avons déjà défini dans le cas de la résolution standard ce qu'est la subsomption. Mais nous donnons ici une définition de subsomption particulière pour cette section qui est plus simple puisqu'elle ne fait pas intervenir de substitution, le système logique n'utilisant pas cet outil.

Définition 5.5.1 (Subsomption) *Une clause C subsume une clause distincte C' si $C \subset C'$.*

Il est possible d'éliminer toute clause subsumée, que celle-ci existe avant ou soit obtenue après la clause qui la subsume :

Proposition 5.5.2 *Soit S un ensemble de clauses, C et C' deux clauses. Soit Γ' une clause. Supposons qu'il existe une dérivation de Γ' à partir de $S; C; C'$ et que C subsume C'. Alors il existe une clause Γ qui subsume Γ' et une dérivation de la clause Γ à partir de $S; C$ dans laquelle la clause C' n'est jamais dérivée et dont le nombre d'utilisation de chacune des clauses de S n'est pas augmenté.*

Preuve : Elle se fait par induction sur le couple (nombre d'utilisation de C' dans la preuve,taille de la preuve). Si C' n'est pas utilisée dans la preuve, on élimine simplement C'. Si C' est utilisée, on peut supposer que la preuve est utile selon 5.3.9 (sinon on la rend utile et la preuve devient plus courte), en ajoutant éventuellement C si celle-ci n'est pas utilisée. Si aucune règle n'est utilisée, c'est que $\Gamma' = C'$. Si l'on pose $\Gamma = C$ alors on a le résultat.
Sinon on peut supposer que la première règle est une règle sur C'. Sinon la première règle R crée une clause D à partir d'une clause de S, et on a une preuve de Γ' à partir de $S; C; C'; D$ qui est plus courte (et le nombre d'utilisation de chacune des clauses est inférieur ou égal). Alors l'induction permet de conclure et comme

$$\frac{S; C}{S; C; D}\,R$$

nous avons bien le résultat souhaité. Le nombre d'utilisation des clauses de S en haut est la somme des utilisations des clauses de S et de D en bas.
Regardons donc la première règle, qui porte sur C'.
– Si c'est une règle quelconque différente de la résolution et de la contraction, $C' = A, C'_1$ et la règle R crée une clause B, C'_1. Il existe donc une dérivation de Γ' à partir de $S; C; C'; B, C'_1$ qui est plus courte. Il y a deux cas :

1. $A \notin C$: alors C subsume aussi B, C'_1 et par induction il existe une preuve d'une clause Γ_1 qui subsume Γ' à partir de $S; C; C'$. L'utilisation de la clause C' est strictement inférieure à précédemment, donc une seconde utilisation de l'hypothèse d'induction

nous donne le fait qu'il existe une clause Γ qui subsume Γ_1 et qui est dérivée à partir de $S; C$.

2. $A \in C = A, C_1$: on peut ajouter la clause B, C_1 à $S; C; C'; B, C_1'$. La clause B, C_1 subsume B, C_1', et par induction il existe une dérivation d'une clause Γ_1 à partir de $S; C; C'; B, C_1$. Dans cette preuve, C' est moins utilisée que dans la première, donc par induction, il existe une dérivation d'une clause Γ à partir de $S; C; B, C_1$. Comme

$$\frac{S; A, C_1}{S; C; B, C_1} R$$

est valide (même pour les règles $\exists_p$ et $\forall_n$ car C, C_1' a été remplacée par B, C_1), le cas est terminé.

– Si la règle est la contraction, alors $C' = A, A, C_1'$ et la clause obtenue est A, C_1'. Il y a ici trois cas :

1. $A \notin C$: C'est la même chose que pour le premier cas de la règle quelconque.
2. $A \in C$ mais $A, A \notin C$: Si $C = A, C_1'$, nous considérons en fait une preuve plus courte de Γ sans utiliser cette règle et l'induction termine.
3. $A, A \in C$: la preuve se fait de même manière que le second cas de la règle quelconque.

– Si la règle est la résolution, alors $C' = A, C_1'$ par exemple (sinon inverser A et $A^\perp$) et il existe une clause $D' = A^\perp, D_1'$ et la clause obtenue est C_1', D_1'. Deux cas ici

1. $A \notin C$: similaire au premier cas de la règle quelconque.
2. $A \in C$: similaire au second cas de la règle quelconque. □

5.6 Le système implémenté

Il est important de noter que ce que nous avons donné au début de ce chapitre est bien un système de démonstration et non une description du fonctionnement du démonstrateur.
Pour être plus précis, les règles où l'on introduit des termes dans le système sont remplacés dans le démonstrateur par des règles où l'on introduit de nouvelles variables, qui auront pour but d'être justement substituées par le bon terme par la suite. Il est bien évident qu'il faut donner une manière de trouver les bons termes à utiliser, et que ceux-ci ne peuvent être trouvés à l'avance.
Les variables du système sont elles remplacées par des constantes fraîches dans le démonstrateur, puisqu'elles ne sont pas substituées. Plus précisément, les variables du système sont en fait remplacées par de nouvelles fonctions ayant pour argument les variables libres de la formule.
Le lemme 5.3.12 nous permet de chercher toutes les règles applicables à une clause, en nombre fini, sans revenir plus tard sur cette clause. En effet, les variables introduites représentant tous les termes possibles, et les constantes introduites, représentant les variables, n'ont besoin de n'être introduites qu'une seule fois grâce au lemme.
Précisons plus cela en donnant le système utilisé pour l'implémentation. Les conventions sont les mêmes que pour le système logique, c'est à dire qu'il s'agit toujours de déductions d'un ensemble de clause à partir d'un ensemble de clause et qu'aucune clause n'est éliminée. Nous notons uniquement les clause actives dans les règles de déduction :

$$\frac{\Gamma, A \vee B}{\Gamma, A, B} \vee_p \quad \frac{\Gamma, (A \vee B)^\perp}{\Gamma, A^\perp} \vee_{n_g} \quad \frac{\Gamma, (A \vee B)^\perp}{\Gamma, B^\perp} \vee_{n_d}$$

$$\frac{\Gamma, A \to B}{\Gamma, A^\perp, B} \to_p \quad \frac{\Gamma, (A \to B)^\perp}{\Gamma, A} \to_{n_g} \quad \frac{\Gamma, (A \to B)^\perp}{\Gamma, B^\perp} \to_{n_d}$$

$$\frac{\Gamma, (A \wedge B)^\perp}{\Gamma, A^\perp, B^\perp} \wedge_n \quad \frac{\Gamma, A \wedge B}{\Gamma, A} \wedge_{p_g} \quad \frac{\Gamma, A \wedge B}{\Gamma, B} \wedge_{p_d}$$

$$\frac{\Gamma, (\neg A)^{\perp}}{\Gamma, A}\,\neg_n \qquad \frac{\Gamma, \neg A}{\Gamma, A^{\perp}}\,\neg_p$$

$$\frac{\Gamma, \forall x.A(x)}{\Gamma, A(y)}\,\forall_p(*) \qquad \frac{\Gamma, (\forall x.A(x))^{\perp}}{\Gamma, A(f(x_1,\ldots,x_n))^{\perp}}\,\forall_n(**)$$

$$\frac{\Gamma, \exists x.A(x)}{\Gamma, A(f(x_1,\ldots,x_n))}\,\exists_p(**) \qquad \frac{\Gamma, (\exists x.A(x))^{\perp}}{\Gamma, A(y)^{\perp}}\,\exists_n(*)$$

$$\frac{\Gamma, A, A'}{\Gamma\sigma, A\sigma}\,contr_p(***) \qquad \frac{\Gamma, A^{\perp}, A'^{\perp}}{\Gamma\sigma, A\sigma^{\perp}}\,contr_n(***)$$

$$\frac{\Gamma, A; \Gamma', A'^{\perp}}{\Gamma\sigma, \Gamma'\sigma}\,res(***)$$

Où

$(*)$ y est une variable fraîche, c'est à dire y n'est pas libre dans S, Γ et $A(x)$;

$(**)$ f est un nom de fonction frais et les x_i sont les variables libres de A sauf x ;

$(***)$ σ est l'unificateur le plus général de A et A'.

Nous considérerons que chaque nouvelle clause subit un renommage des variables de telle sorte que ses variables libres soient toutes des variables fraîches. Ainsi l'unification lors de la résolution se fait sur des formules ayant des variables distinctes.

Remarque 5.6.1 Les règles sont les mêmes que le système logique pour les cas propositionnels, mais changent de manière importante pour les règles portant sur les quantificateurs, les contractions et la résolution.

5.7 Complétude du système implémenté

Nous souhaitons prouver que ce système est complet, et pour cela nous allons montrer qu'il est équivalent au calcul des séquents, d'une manière similaire au système logique. Nous devons tout d'abord rappeler un résultat sur les fonctions de Skolem. Pour cela, rappelons la sémantique du calcul des séquents :

Définition 5.7.1 *La validité d'un séquent est définie comme suit :*

- *Un séquent $\Gamma \vdash \Delta$ exprimé dans le langage $\mathcal{L}$ est valide dans une $\mathcal{L}$-structure $\mathcal{M}$ si pour toute substitution σ de ses variables libres par des éléments de $\mathcal{M}$ la validité de toutes les formules de $\Gamma\sigma$ dans $\mathcal{M}$ implique la validité d'au moins une formule de $\Delta\sigma$ dans $\mathcal{M}$.*
- *Un séquent est dit valide si il est valide dans toute $\mathcal{L}$-structure.*
- *Soit Γ un ensemble de formules. On dit que Γ est contradictoire si $\Gamma \vdash$ est valide.*

Remarque 5.7.2 Le fait qu'un ensemble de formules Γ soit contradictoire signifie que quelle que soit la structure, et quelle que soit la substitution σ la conjonction des formules de $\Gamma\sigma$ n'est pas valide dans la structure. Sinon comme le séquent $\Gamma \vdash$ est valide il doit y avoir une formule à droite du séquent dont on peut tester la validité, ce qui est impossible.

On remarque que si le séquent n'a pas de variable libre alors la définition peut être simplifiée, il est inutile de parler de substitution.
On a de plus le théorème de complétude suivant que l'on rappelle :

Proposition 5.7.3 *Un séquent est démontrable en calcul des séquents sans coupure si et seulement si il est valide.*

Nous pouvons maintenant anoncer et prouver le lemme suivant :

Lemme 5.7.4 *Soit S un ensemble de formules closes, soit G et H des formules. Supposons que G ait $x, x_1, \ldots, x_n$ comme variables libres et que H ait $y_1, \ldots, y_m$ comme variables libres supplémentaires. Soit f un nouveau symbole de fonction à n arguments. Alors on a l'équivalence de*

$$S, \forall x_i, y_i(H \vee \exists x.G(x, x_1, \ldots, x_n)) \vdash_{LK}$$

avec

$$S, \forall x_i, y_i(H \vee G(f(x_1, \ldots, x_n), x_1, \ldots, x_n)) \vdash_{LK}$$

Preuve : Notons pour la suite $F = \exists x.G(x, x_1, \ldots, x_n)$.

- prouvons le sens ($\Rightarrow$) par la contraposée. Supposons donc qu'il existe une structure telle que toutes les formules de $S, \forall x_i, y_j(H \vee G(f(x_1, \ldots, x_n), x_1, \ldots, x_n))$ sont valides dans la structure. Il suffit de prouver que $\forall x_i, y_j(H \vee F)$ est valide dans cette structure. Soit σ une substitution ayant pour domaine les variables x_i et y_j et pour image des éléments de la structure. Alors $(H \vee G(f(x_1, \ldots, x_n), x_1, \ldots, x_n))\sigma$ est valide.
  Nous devons prouver que $(H \vee F)\sigma$ est valide. Si $H\sigma$ est satisfaite, alors c'est terminé. Dans le cas contraire, c'est $G(f(x_1, \ldots, x_n), x_1, \ldots, x_n)\sigma$ qui est valide, donc il y a un terme $t = f(x_1, \ldots, x_n)\sigma$, qui est tel que $G(x, x_1, \ldots, x_n)\sigma[t/x]$ est satisfaite. Donc $F\sigma$ est satisfaite, puis $(H \vee F)\sigma$ également.
- de même dans l'autre sens, supposons qu'il y ait une structure telle que les formules de $S, \forall x_i, y_j(H \vee F)$ soient valides dans la structure. Il suffit d'étendre la structure en donnant une interprétation de f pour laquelle
  $\forall x_i, y_j.(H \vee G(f(x_1, \ldots, x_n), x_1, \ldots, x_n))$ est valide.
  Soit σ_x une substitution transformant les variables x_i en des éléments de la structure et notons σ_y toute substitution transformant les variables y_j en des éléments de la structure. Si $H\sigma_x\sigma_y$ est valide pour toute σ_y, alors on interprète f en $(x_1, \ldots, x_i)\sigma_x$ par n'importe quel élément de la structure.
  Sinon fixons une substitution σ_y pour laquelle $H\sigma_x\sigma_y$ n'est pas valide. On sait que $F\sigma_x\sigma_y$ est valide, et par conséquent il existe un élément t de la structure tel que
  $G(x, x_1, \ldots, x_n)\sigma[t/x]$ est valide. On interprète alors f en $(x_1, \ldots, x_i)\sigma_x$ par cet élement t.
  Nous avons défini entièrement l'interprétation de f telle façon que
  $\forall x_i, y_j(H \vee G(f(x_1, \ldots, x_n), x_1, \ldots, x_n))$ est valide. □

Lemme 5.7.5 *Soit S un ensemble de formules closes, soit G et H des formules. Supposons que G ait $x, x_1, \ldots, x_n$ comme variables libres et que H ait $y_1, \ldots, y_m$ comme variables libres supplémentaires. Soit f un nouveau symbole de fonction à n arguments. Alors*

$$S, \forall x_i, y_i(H \vee \neg\forall x.G(x, x_1, \ldots, x_n)) \vdash_{LK}$$

si et seulement si

$$S, \forall x_i, y_i(H \vee \neg G(f(x_1, \ldots, x_n), x_1, \ldots, x_n)) \vdash_{LK}$$

Preuve : Le fait que dans toute structure $\neg\forall x.F$ soit équivalent à $\exists x.\neg F$ nous permet de conclure en utilisant le lemme précédent. □

Nous avons besoin de quelques notations supplémentaires pour continuer.

Définition 5.7.6 *Soit $\Gamma^\perp, \Delta$ une clause non vide. Alors on définit*

$$\phi(\Gamma^\perp, \Delta) = \forall x_1, \ldots, x_n(\vee_i \neg G_i) \vee (\vee_j H_j)$$

où les x_i sont les variables libres de la clause.
Si S est un ensemble de clauses $\{\Gamma_k^\perp, \Delta_k\}_{k=1..n}$ nous notons $\phi(S)$ l'ensemble des images des clauses de S par ϕ :

$$\phi(S) = \{\phi(\Gamma_k^\perp, \Delta_k)\}_{k=1..n}$$

Remarque 5.7.7 la fonction ϕ appliquée à une clause renvoie la formule la représentant.

5.7.1 Le système implémenté est sûr

Proposition 5.7.8 *Soit $S = \{\Gamma_k^{\perp}, \Delta_k\}_{k=1..n}$ un ensemble de clauses non vides. Supposons que l'on puisse dériver la clause vide à partir de S. Alors $\phi(S) \vdash_{LK}$.*

Preuve : Celle-ci se fait par induction sur la longueur de la dérivation de S à la clause vide. Comme S ne contient pas la clause vide, il y a nécessairement une règle qui est utilisée. C'est à dire que la dérivation ressemble à

$$\dfrac{\dfrac{\dfrac{S}{S;C}\,R}{\vdots}}{\square}$$

Si C est la clause vide, cela signifie que R est la règle de résolution, et qu'il existe des formules F et F' qui s'unifient et telles que F et $F'^{\perp}$ soient des clauses de S. Alors S peut s'écrire $S = S'; F; F'^{\perp}$. Nous voulons alors prouver que $\phi(S'), \forall x_i F, \forall y_j \neg F' \vdash_{LK}$. Soit σ l'unification qui rend F et F' égales.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{F\sigma \vdash F'\sigma}\,ax}{F\sigma, \neg F'\sigma \vdash}\,\neg_g}{\phi(S'), F\sigma, \neg F'\sigma \vdash}\,w_g^*}{\phi(S'), \forall x_i F, \forall y_j \neg F' \vdash}\,\forall_g^*$$

Si C n'est pas vide, alors $S; C$ est un ensemble de clauses non vides à partir duquel on peut dériver la clause vide avec une longueur de dérivation plus courte que pour S. Ainsi l'hypothèse d'induction nous permet de conclure que $\phi(S), \phi(C) \vdash_{LK}$. Nous devons prouver que de $\phi(S), \phi(C) \vdash$ on peut déduire $\phi(S) \vdash$. Pour cela il suffit de prouver que

– dans le cas où la règle n'a qu'une formule active $\phi(D_1) \vdash_{LK} \phi(C)$ si C a été obtenu à partir de la clause D_1 de $S = S'; D_1$ car alors :

$$\dfrac{\dfrac{\phi(S'), \phi(D_1), \phi(C) \vdash \quad \phi(D_1) \vdash \phi(C)}{\phi(S'), \phi(D_1), \phi(D_1) \vdash}\,cut}{\phi(S) \vdash}\,c_g^*$$

– pour le cas où la règle a deux formules actives $\phi(D_1), \phi(D_2) \vdash_{LK} \phi(C)$ si C a été obtenu à partir de la clause D_1 et de la clause D_2 de $S = S'; D_1; D_2$ car alors :

$$\dfrac{\dfrac{\phi(S'), \phi(D_1), \phi(D_2), \phi(C) \vdash \quad \phi(D_1), \phi(D_2) \vdash \phi(C)}{\phi(S'), \phi(D_1), \phi(D_2), \phi(D_1), \phi(D_2) \vdash}\,cut}{\phi(S) \vdash}\,c_g^*$$

Nous noterons Π la formule $\phi(\Gamma)$ sans lier les variables libres pour Γ une clause et $\Sigma = \phi(S')$ pour S' un ensemble de clauses. Voyons donc les différents cas selon la règle R. Nous donnons des preuves sémantiques. Tous les cas ne seront pas donnés ici et le lecteur intéressé pourra aller voir le reste de la preuve à l'annexe V :

– $R = \forall_p$: S contient une clause $\Gamma, \forall x.A(x)$ et $C = \Gamma, A(z)$ avec z une variable fraîche. Nous devons donc prouver $\forall x_i(\Pi \vee \forall x.A(x)) \vdash_{LK} \forall x_i, z(\Pi \vee A(z))$
Si la première formule est valide dans une structure, prouvons que la seconde est valide. Si elle ne l'est pas, cela signifie qu'il y a une substitution σ des variables pour laquelle $(\Pi \vee A(z))\sigma$ est non valide, ce qui signifie que $\Pi\sigma$ est non valide et $A(z)\sigma$ est non valide. Or si $\Pi\sigma$ est non valide, alors par hypothèse $\forall x.A(x)\sigma$ est valide, ce qui signifie que $A(z)\sigma$ en particulier est valide, ce qui est contradictoire.

– $R = \forall_n : S$ contient une clause $\Gamma, (\forall x.A(x, x_1, \ldots, x_n))^\perp$ et une clause
$C = \Gamma, A(f(y_1, \ldots, y_n), y_1, \ldots, y_n)^\perp$ avec f une nouvelle fonction et y_k les variables libres de $A(y, y_1, \ldots, y_n)$ en dehors de y. Nous avons par hypothèse d'induction que
$\Sigma, \forall x_i(\Pi \vee \neg\forall x.A(x, x_1, \ldots, x_n)), \forall y_j \Pi \vee \neg A(f(y_1, \ldots, y_n), y_1, \ldots, y_n) \vdash_{LK}$ (1)
Nous souhaitons prouver $\Sigma, \forall x_i(\Pi \vee \neg\forall x.A(x, x_1, \ldots, x_n)) \vdash_{LK}$.
Or par le lemme 5.7.5 (1) est équivalent à
$\Sigma, \forall x_i(\Pi \vee \neg\forall x.A(x, x_1, \ldots, x_n)), \forall y_i(\Pi \vee \neg\forall y.A(y, y_1, \ldots, y_n)) \vdash_{LK}$
Ce qui est équivalent au résultat (par contraction dans un sens et affaiblissement dans l'autre).
– voir les autres cas dans l'annexe V. □

5.7.2 Le système implémenté est complet

Nous avons besoin du résultat sur la résolution standard, qui consiste à utiliser uniquement les règles de résolution et de contraction sur des clauses dont les éléments sont des formules atomiques.

Théorème 5.7.9 (Complétude de la résolution standard) *Soit S un ensemble de clauses de formules atomiques. Alors $\phi(S)$ est contradictoire si et seulement si il existe une dérivation de la clause vide à partir de S en utilisant les règles de résolution et de contraction.*

Définition 5.7.10 *Dans cette section nous définissons la taille $t(F)$ d'une formule F par induction ainsi :*
– $t(Atome) = 0$;
– $t(F_1 @ F_2) = 1 + t(F_1) + t(F_2)$ si $@$ est un opérateur binaire ;
– $t(*F) = 1 + t(F)$ si $*$ est la négation ou un quantificateur.
On définit également la taille d'une formule d'une clause par $t(A^\perp) = t(A)$.

Nous allons utiliser le lemme ci-dessous de manière légèrement cachée dans la preuve de la proposition 5.7.12 :

Lemme 5.7.11 *Soit S un ensemble de clauses et $C = A, A, \Gamma$ une clause. Alors $\phi(S; C)$ est contradictoire si et seulement si $\phi(S; A, \Gamma)$ est contradictoire.*

Preuve : Il suffit de prouver que $\phi(A, A, \Gamma)$ est équivalent à $\phi(A, \Gamma)$, ce qui est clair, car $A \vee A$ est équivalent à A. □

Proposition 5.7.12 *Soit S un ensemble de clauses. Si $\phi(S)$ est contradictoire alors on peut dériver la clause vide à partir de S.*

Preuve : Elle se fait par induction sur le couple (taille maximale des formules de S, nombre de formules distinctes de taille maximale), en considérant que deux formules sont égales à renommage des variables près. Si la taille maximale des formules est nulle, cela signifie que toutes les formules sont atomiques et par conséquent le théorème de complétude de la résolution standard nous permet de conclure.
Sinon choisissons une formule F de taille maximale (non nulle) d'une clause de S. Notons $C_i = C_i', F$ les clauses de $S = S'; C_i$. Si F apparaît plusieurs fois dans C_i, nous considérons qu'elle n'apparaît qu'une seule fois grâce au lemme précédent. Regardons les cas selon F, et donnons un ensemble T de clauses tel que $\phi(T)$ est contradictoire et dont le couple est plus petit. Pour cela il suffit de prendre toutes les décompositions de F, qui sont toujours de taille inférieure, et d'éliminer la formule F. Nous ne donnons pas ici tous les cas possibles. Le lecteur désirant voir les cas manquant pourra se reporter à l'annexe V :
– $F = \forall x.A(x)$: Posons $T = S'; C_i', A(z)$ avec z une nouvelle variable. Si $\phi(T)$ n'est pas contradictoire, alors il existe une structure dans laquelle toutes les formules de $\phi(T)$ sont valides. Donc les formules de $\phi(S')$ sont valides et $\phi(C_i', A(z)) = \forall y_i, z(C_i' \vee A(z))$ également.

Or au moins une des formules $\phi(C_i)$ n'est pas valide. Donc il existe une substitution σ telle que $C_i'\sigma \vee \forall x.A(x)\sigma$ n'est pas valide. donc $C_i'\sigma$ est non valide et $\forall x.A(x)\sigma$ est non valide. En particulier $A(z)\sigma$ est non valide donc $\phi(C_i', A(z))$ n'est pas valide : contradiction.
- $F = (\forall x.A(x, x_1, \ldots, x_n))^\perp$: Posons $T = S'; C_i', A(f(x_i), x_1, \ldots, x_n)^\perp$ avec f un nouveau symbole de fonction, et x_i les variables libres de A en dehors de x. Nous savons par le lemme 5.7.5 que $\phi(T)$ est contradictoire si et seulement si $\phi(S)$ est contradictoire.
- voir les autres cas dans l'annexe V.

On remarque que les clauses ajoutées à S' pour créer T peuvent toujours être obtenues à partir de règles sur les clauses C_i, en décomposant F après avoir fait une contraction dans le cas où la formule F apparaît plusieurs fois dans la clause. Autrement dit, S se déduit en $T; C_i$, et puisque l'on peut déduire la clause vide à partir de T par hypothèse d'induction, il en est de même pour S. □

Remarque 5.7.13 Soit S un ensemble de clauses. Nous avons donc maintenant grâce aux propositions 5.7.8 et 5.7.12 l'équivalence entre
- On peut déduire la clause vide de S ;
- $\phi(S)$ est contradictoire.

On remarque que prouver qu'une formule close F est valide est équivalent à prouver que $\neg F$ est contradictoire, donc que l'on peut déduire la clause vide à partir de la clause $F^\perp$. Dans le cas d'une formule non close, il faut considérer les variables comme des constantes :

$$\vdash F(x) \Leftrightarrow \vdash \forall x.F(x) \Leftrightarrow \neg\forall x.F(x) \vdash \Leftrightarrow \neg F(c) \vdash$$

Définition 5.7.14 *Nous dirons par abus dans la suite que S est contradictoire si $\phi(S)$ est contradictoire. Il s'agit bien d'un abus, car $F(x), \neg F(y) \vdash$ n'est pas dérivable, alors que $\forall x.F(x), \forall y.\neg F(y) \vdash_{LK}$. En effet pour le premier il existe une structure et une substitution pour lesquelles les deux formules sont valides : il suffit de prendre une structure dans laquelle il y a deux constantes a et b telles que $F(a)$ est valide et $F(b)$ est non valide. Pour le second, la preuve évidente est laissée au lecteur.*

Remarque 5.7.15 Ce que prouve la proposition 5.7.12, c'est qu'il est possible de faire une preuve où l'on décompose toutes les formules pour aboutir à des clauses contenant uniquement des formules atomiques, sur lesquelles on peut effectuer la résolution standard. Seulement le système est fait de telle façon qu'il n'est justement pas obligatoire d'opérer ainsi. Par exemple, si l'on cherche à prouver la formule $F \to F$, et que F est une formule quelconque, aussi complexe qu'on le souhaite, alors la preuve suivante ne décompose pas F :

$$\dfrac{\dfrac{\dfrac{(F \to F)^\perp}{(F \to F)^\perp; F}\to_{n_g}}{(F \to F)^\perp; F; F^\perp}\to_{n_d}}{\square}\, res$$

5.8 Une stratégie complète du système implémenté

A la manière de la preuve de la complétude du système implémenté, nous pouvons faire une preuve pour une stratégie complète : l'élimination des clauses subsumées et des tautologies. Remarquons que pour le système logique nous n'avons pas ajouté l'élimination des tautologies à notre stratégie. La recherche d'une telle preuve a en fait abouti à la nécessité de faire la preuve pour le système implémenté.
En effet les règles sur les quantificateurs ne permettent pas l'élimination de la formule parent pour le système logique comme c'est le cas pour le système implémenté.
Or le fait de pouvoir éliminer les clauses décomposées permet comme nous l'avons vu dans la preuve de complétude de faire une preuve par induction en faisant décroitre la taille des formules.

L'exemple ci-dessous permet d'illustrer la raison pour laquelle il est impossible d'éliminer les clauses décomposées dans le système logique.

Exemple 5.8.1 Etudions la formule $\exists x.\forall y.(F(x) \to F(y))$. La preuve dans les deux systèmes est différente. Donnons-les d'une manière simplifiée, sans donner toutes les clauses (aucune n'est éliminée).
Dans le système logique :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\neg\exists x.\forall y.(F(x) \to F(y))}{\neg\forall y.(F(a) \to F(y))}\exists_n}{\neg(F(a) \to F(y))}\forall_n}{\neg F(y)}\to_{n_d}}{\neg\forall z.(F(y) \to F(z))}\exists_n}{\neg(F(y) \to F(z))}\forall_n}{F(y)}\to_{n_g}}{\square}\, res$$

Dans le système implémenté :

$$\dfrac{\dfrac{\dfrac{\dfrac{\neg\exists x.\forall y.(F(x) \to F(y))}{\neg\forall y.(F(x) \to F(y))}\exists_n}{\neg(F(x) \to F(f(x)))}\forall_n}{F(x); \neg F(f(x))}\to_{n_g}; \to_{n_d}}{\square}\, res$$

La preuve est plus courte dans le second système que dans le premier. Cela vient du fait que la variable y dans le système logique doit être utilisée par une seconde utilisation de la règle $\exists_n$. Elle ne peut pas être utilisée avant à cause du fait que lors de l'application de la règle $\forall_n$, la variable doit être fraîche. Cela montre bien qu'il n'est pas possible d'éliminer les clauses précédentes. Dans le système implémenté $F(x)$ et $\neg F(f(x))$ sont deux clauses distinctes et donc leurs variables sont distinctes également ce qui permet d'appliquer la règle de résolution.

5.8.1 L'élimination des clauses subsumées et des tautologies

Il s'agit d'une stratégie standard, qui consiste à ne pas créer de tautologies, ni de clauses subsumées par des clauses anciennes (subsomption avant), et à éliminer les clauses subsumées par une nouvelle clause (subsomption arrière). Nous donnons les définitions de subsomption et de tautologie. Ces définitions ne sont valables que pour cette section.

Définition 5.8.2 *Une clause C subsume une clause C' si il existe une substitution σ n'unifiant aucune des formules de C telle que $C\sigma \subset C'$.*

On remarque qu'il s'agit de la même définition que dans le chapitre 4 à la section 4.3.1, en ayant ajouté la propriété nécessaire à la complétude de stratégie d'élimination.

Définition 5.8.3 *Une clause C est une tautologie si elle contient une formule F et sa négation $F^{\perp}$.*

Pour prouver la complétude de la stratégie, nous avons besoin du théorème usuel sur la résolution standard que nous donnons sous la forme suivante :

Théorème 5.8.4 (Complétude en résolution standard) *Soit S un ensemble de clauses de formules atomiques. Alors $\phi(S)$ est contradictoire si et seulement si il existe une dérivation de*

la clause vide à partir de S en utilisant les règles de résolution et de contraction, sans créer de clauses subsumées ni de tautologies et en éliminant à chaque étape les clauses subsumées par la clause nouvelle à chaque étape.

Nous avons besoin de quelques lemmes avant de prouver le résultat principal. Ces lemmes sont usuels également, mais comme leur preuve est courte, nous pouvons les redonner ici.

Lemme 5.8.5 *Soit S un ensemble de clauses, C et C' deux clauses telles que C subsume C'. Alors $S; C, C'$ est contradictoire si et seulement si $S; C$ est contradictoire.*

Preuve : Si $\phi(S; C)$ est contradictoire, alors $\phi(S; C; C')$ l'est aussi. En effet s'il y a une structure dans laquelle toutes les formules de $\phi(S; C; C')$ sont valides, alors dans cette structure toutes les formules de $\phi(S; C)$ sont valides.
Réciproquement, supposons l'existence d'une structure dans laquelle toutes les formules de $\phi(S; C)$ sont valides. En particulier $\phi(C)$ est valide. On sait par hypothèse qu'il existe une substitution σ et une clause C_1 telles que $C' = C\sigma, C_1$. Supposons que $\phi(C)$ que l'on note $\forall x_1 \dots x_n.F(x_1, \dots, x_n)$ soit valide, il suffit de prouver que
$\phi(C') = \forall y_1 \dots y_m.F(x_1, \dots, x_n)\sigma \vee G(y_1, \dots, y_m)$, où σ est une substitution qui renomme les variables libres x_i de F, est valide. Soit y_1,...,y_n, il suffit de prouver que $F(x_1, \dots, x_n)\sigma$, ce qui est le cas puisque pour voir cela on applique $\phi(C)$ avec les termes $\sigma(x_i)$. □

Lemme 5.8.6 *Soit S un ensemble de clauses et C une clause. Supposons que C soit une tautologie. Alors $S; C$ est contradictoire si et seulement si S est contradictoire.*

Preuve : Comme précédemment, si S est contradictoire, alors $S; C$ l'est également. Réciproquement, supposons l'existence d'une structure dans laquelle toutes les formules de $\phi(S)$ sont valides. Nous devons prouver que $\phi(C)$ est valide. Or C est une tautologie, donc $\phi(C) = \forall x_1 \dots x_n.F \vee \neg F \vee G$. Il est clair que $\phi(C)$ est valide dans toute structure car $F \vee \neg F$ est toujours valide. □

Proposition 5.8.7 *Soit S un ensemble de clauses. Si S est contradictoire, alors on peut dériver la clause vide à partir de S sans créer de clauses subsumées ni de tautologies et en éliminant à chaque étape les clauses subsumées par la nouvelle clause.*

Preuve : Elle se fait par induction sur le couple (taille maximale des formules de S, nombre de formules distinctes de taille maximale). Nous pouvons supposer grâce aux lemmes précédents que S ne contient pas de tautologie, ni de clause subsumée. Si la somme des tailles des formules est nulle, alors le théorème 5.8.4 nous permet de conclure.
Sinon, choisissons une formule de taille maximale, et décomposons là dans toutes les clauses dans laquelle elle apparaît après une éventuelle contraction. Nous savons que nous pouvons ajouter toutes ces décompositions à S et enlever les clauses contenant F tout en conservant la propriété que l'ensemble de clauses est contradictoire, ceci d'après la preuve que nous avons faite de la complétude du système implémenté.
Si dans ces décompositions il y a des tautologies, nous pouvons ne pas les ajouter, de même s'il y a des clauses subsumées par des clauses anciennes. Nous conservons en effet un ensemble contradictoire grâce aux lemmes précédents. Nous pouvons effectuer l'élimination des clauses anciennes qui sont subsumées par les nouvelles en conservant un ensemble de clauses contradictoires.
Ce que nous obtenons est alors un ensemble de clauses contradictoire pour lequel le couple a diminué puisque F n'apparaît plus. Nous appliquons alors l'induction qui nous permet de conclure. □

Perspectives

À ce jour, le projet n'a pas abouti à un prototype permettant l'analyse complète d'une démonstration. Comme nous avons pu le voir, les traductions de preuve écrites en langue naturelle dans le langage restreint restent pour le moment manuelles.
Ce projet porte cependant sur un sujet enthousiasmant, et on peut tout à fait imaginer qu'un tel système verra le jour. Il est à espérer que les idées développées au long de ces trois années soient utilisées pour la réalisation d'un projet semblable.
Bien que ce projet se termine, il reste néanmoins des développements intéressants à traiter. Plusieurs points sont à explorer, et cela dans la plupart des parties étudiées dans cette thèse :
• d'un point de vue pratique :

1. continuer à étendre le langage restreint. Celui-ci reste incomplet malgré une amélioration apportée par rapport à la première version introduite par Christophe Raffalli avec les new_commands :
   - nous avons homogénéisé le langage, dans le sens où certaines commandes comme *prove* n'étaient pas considérées comme des commandes du langage, mais comme une commande primitive de l'assistant de démonstration PhoX ;
   - nous avons ajouté la possibilité de décrire un arbre de preuve et non pas seulement des règles isolées ;
   - nous avons également donné une sémantique plus complexe, avec une interprétation plus fine permettant de simplifier les buts donnés au démonstrateur automatique.

   Il manque cependant des possibilités :
   - permettre au démonstrateur de donner une valeur aux variables globales. Nous avons cependant donné une piste permettant de faire cela ;
   - permettre les preuves par induction, qui devraient être autorisées par un langage plus développé.
2. optimiser le démonstrateur. Un démonstrateur n'est jamais terminé, car il y a toujours des erreurs à corriger, et des améliorations à faire :
   - la chose la plus importante dans notre démonstrateur est le calcul des poids. Celui-ci doit vraiment être optimisé pour que les preuves soient les plus courtes et les plus rapides possibles. Un des moyens possibles pour cette optimisation serait l'utilisation d'un algorithme génétique, modifiant des paramètres issus des formules calculant les poids. Cela permettrait de trouver les valeurs des paramètres rendant le temps de calcul de la machine le plus court possible sur une liste de problèmes donnée.
   - la gestion des utilisations d'hypothèses, indiquées par le langage restreint, est également un point à développer. Celle-ci n'en est qu'à son début, et trop peu d'exemples à ce jour permettent de faire un réel choix pour la méthode à employer.
   - améliorer les affichages, que ce soit le mode verbeux lors de l'affichage des preuves, ou les messages d'erreur affichés après l'analyse des commandes. Ces deux choses sont distinctes mais relèvent du même problème de lisibilité et de convivialité qui manquent jusqu'à présent.
   - le système actuel ignore le raisonnement équationnel. Dans le cas de l'interface avec PhoX il faut que les égalités qui apparaissent pendant la preuve soient des hypothèses pour qu'elles soient utilisées dans un raisonnement équationnel lors des

unifications. Cela empêche alors certaines étapes de preuves un peu longues. Utiliser la paramodulation pourait être une solution.
- implémenter un système permettant de vérifier la preuve donnée par le démonstrateur, et ceci de manière indépendante. Jusqu'à présent la seule vérification qui peut être faite est la lecture de la preuve affichée. Or il pourrait être plus intéressant d'avoir un programme externe au démonstrateur qui puisse valider la preuve donnée automatiquement.

- d'un point de vue théorique :
  1. La preuve faite dans le chapitre 3 sur le type des termes η-long est assez complexe, et nous souhaiterions trouver plus simple. Une piste suggérée serait de définir un calcul semblable, mais avec du sous-typage. Il peut en effet être intéressant de voir une généralisation du calcul, avec éventuellement plusieurs types de flèches.
  2. Dans le λ-calcul avec deux flèches, nous souhaiterions étudier le problème d'unification ou de filtrage. Comme nous l'avons déjà signalé, il existe un λ-calcul (cf. [CePfe1]) dans lequel le problème d'unification a été traité (cf. [CePfe2]). Cependant ce système a deux types d'application, ce qui simplifie un peu les choses, en particulier pour le typage des éléments. Rappelons que dans notre cas il n'est pas fait de distinction entre une application linéaire et une application intuitionniste. Il pourrait donc être intéressant de traiter de ce problème particulier.
  3. il peut être également intéressant d'étudier d'autres extensions des ACGs, permettant une utilisation plus agréable de celles-ci. Une extension possible serait l'ajout de traits aux types atomiques qui permettent de donner des précisions sur les éléments dans les signatures. Par exemple, pour le type nom, nous pouvons ajouter les traits correspondant au nombre (singulier, pluriel) et au genre (masculin, féminin). Cela peut permettre de réduire le nombre des entrées dans les signatures, dans le cas où certains éléments peuvent avoir des traits indifférents il serait possible de les factoriser dans une seule entrée.
  4. l'étude du système logique théorique doit être continuée. Le point le plus intéressant est la recherche de stratégies complètes. En particulier une stratégie parlant de décomposition minimum des formules serait très utile. S'agissant du but même du démonstrateur, nous aimerions prouver qu'il existe une manière d'empêcher certaines résolutions dans le cas où trop de décompositions auraient été faites.

     Une stratégie possible semble être celle qui interdit la résolution entre deux clauses si les formules unifiables sont "cousines", c'est à dire proviennent par décomposition de formules unifiables. Pour être plus précis, si A, Γ et $\neg A', \Gamma'$ sont telles que A provient d'une formule F (resp. $\neg F$) et $\neg A'$ provient d'une formule $\neg F'$ (resp. F') et si F et F' sont unifiables, alors il la résolution entre A, Γ et $\neg A', \Gamma'$ est interdite.

Bibliographie

[Ab] P. W. Abrahams, *Machine verification of mathematical proofs*, PhD thesis, MIT, 1963

[Alfa] *The proof editor Alfa*, `http://www.cs.chalmers.se/~hallgren/Alfa/`

[AsBuVi] N. Asher, J. Busquets, L. Vieu, *La SDRT : Une approche de la cohérence du discours dans la traduction sémantique dynamique*

[As] N. Asher, *Reference to abstract objects in discourse : A Philosophical Semantics for Natural Language Metaphysics and Epistemology*, Kluwer, Dordrecht, 1993

[Auto] *Automath Archive*, `http://www.win.tue.nl/automath/`

[BaGa] L. Bachmair, H. Ganzinger, *Resolution theorem prover*, in Handbook of automated reasonning, chapter 2, pages 19-99, North-Holland, 2001

[BeKaThe] Y. Bertot, G. Kahn, L. Théry, *Proof by Pointing*, Symposium on Theoretical Aspects Computer Software (STACS), Sendai (Japan), LNCS 789, April 1994

[Bru] K. Brünnler, *Atomic Cut Elimination for Classical Logic*, Lecture Notes in Computer Science, CSL 2003 vol 2803 pp 86-97, 2003

[BruTi] K. Brünnler, A. F. Tiu *A Local System for Classical Logic*, R. Nieuwenhuis and A. Voronkov, editors, LPAR 2001

[CaCo] F.Cardone et M. Coppo, *Two extensions of Curry's types inference system*, Logic and science computer science, v. 31, pp 19-75, 1990

[CePfe1] I. Cervesato et F. Pfenning, *A linear Logical Framework*, 11th Annual Symposium on Logic in Computer Science - LICS'96 (E. Clarke, editor), pp. 264-275, IEEE Computer Society Press, New Brunswick, NJ, 27-30 July 1996

[CePfe2] I. Cervesato et F. Pfenning, *Linear Higher-Order Pre-Unification*, International Workshop on Proof-Search in Type-Theoretic Languages - PSTT'96 (D. Galmiche, editor), pp. 41-50, New Brunswick, NJ, 30 July 1996

[ChaMaPa] E. Chailloux, P. Manoury et B. Pagano, *Développement d'applications avec Objective Caml*, O'Reilly, 2000

[ChaLe] C.-L. Chang, R. C.-T. Lee, *Symbolic Logic and mechanical Theorem proving*, Academic Press, 1973

[CoKo] J. Courtin et I. Kowarski, *Initiation à l'algorithmique et aux structures de données Volume 2*, Dunod, 1995

[Coq] *L'outil d'aide à la preuve Coq*, `http://coq.inria.fr/coq-fra.html`

[Co] Y. Coscoy, *Explication textuelles de preuves pour le calcul des constructions inductives*, Université de Nice-Sophia-Antipolis, Thèse d'université, 2000

[DaMi] L. Damas et R. Milner, *Principal type-schemes for functional programs*, 9th symposium Principles of programming Languages, pp 207-212. ACM Press, 1982

[DaNoRa] R. David, K. Nour, C. Raffalli, *Introduction à la logique*, Dunod, 2001

[DeG] P. de Groote, *Towards Abstract Categorial Grammars*, in Association for Computational Linguistics, 39th Annual Meeting and 10th Conference of the European Chapter, Proceedings of the Conference, pp. 148–155, 2001

[Du] D.A. Duffy, *Principles of automated theorem proving*, Wiley, 1991

[ETPS] P.B. Andrews, M. Bishop, C. E. Brown, S. Issar, F. Pfenning, H. Xi,*ETPS : A System to Help Students Write Formal Proofs*, Journal of Automated Reasoning 32, 75-92, 2004

[Fi] M. Fitting, *First-order Logic and automated Theorem proving*, Springer-Verlag, 1990

[GeNeVri] J.H. Geuvers, R.P. Nederpelt, R.C. de Vrijer, *Selected Papers on Automath*, North-Holland, Amsterdam, 1994

[GiWi] M. Giero, F. Wiedijk, *MMode, a Mizar Mode for the proof assistant Coq*, 2003

[HaRa] T. Hallgren, A. Ranta, *An Extensible Proof Text Editor*, M. Parigot and A. Voronkov (eds), Logic for Programming and Automated Reasoning (LPAR'2000), LNCS/LNAI 1955, pp. 70-84, Springer Verlag, Heidelberg, 2000

[He] M. Hess, *Recent Developments in Discourse Representation Theory*, Communication with Men and Machines, M. King ed., 1991

[Hi] J.R. Hinley, *The principle type scheme of an object in combinatory logic*, Transactions of the American Math Society, v. 146, pp. 29-60, 1969

[HOL] *HOL*, `http://hol.sourceforge.net/`

[Hu] G.P. Huet, *A unification algorithm for typed λ-calculus*, in Theoretical Computer Science 1 (1975) 27-57, North-Holland Publishing Company

[Isa] *Isabelle*, `http://www.cl.cam.ac.uk/research/hvg/Isabelle/`

[Le1] A. Leitsch, *The Resolution Calculus*, Springer-Verlag, 1997

[Le2] A. Leitsch, *Resolution Calculus and Proof Complexity (bis)*, 2nd International Summer School in Logic for Computer Science July 4-15, 1994

[LEGO] *The LEGO Proof Assistant*, `http://www.dcs.ed.ac.uk/home/lego/`

[LeWe] X. Leroy et P. Weis, *Le langage Caml*, Dunod, 1999

[LyPaVe] A. Lyaletski, A. Paskevich, K. Verchinine, *SAD as a mathematical assistant - how should we go from here to there ?*, in Journal of Applied Logic 4, pp 560-591, 2006

[Me] K. Mehlhorn, *Data structures and algoritms 1 : Sorting and searching*, Springer-Verlag, 1984

[Mi] G. Mints, *Inverse Method and Resolution Strategies (bis)*, 2nd International Summer School in Logic for Computer Science July 4-15, 1994

[Mizar] *Mizar project*, `http://www.mizar.org/project/`

[Ne] M. Newborn, *Automated Theorem Proving*, Springer-Verlag, 2001

[NiPa] T. Nipkow, L. C. Paulson, *Isabelle-91 (system abstract)*, D. Kapur (editor), 11th International Conf. on Automated Deduction (Springer LNAI 607), 673-676, 1992

[Pa] M. Parigot, *Free Deduction : An Analysis of "Computations" in Classical Logic*, Lecture Notes in Computer Science 592, 1991

[Pe] F.J. Pelletier, *Seventy-five problems for testing automatic theorem provers*, in Journal of automated reasoning 2 pp 191-216, D. Reidel Publishing Company, 1986

[PhoX] *The PhoX Proof Assistant*,
`http://www.lama.univ-savoie.fr/~raffalli/phox.html`

[Po] R. Pollack, *The Theory of LEGO : A Proof Checker for the Extended Calculus of Constructions*, PhD thesis, Univ. of Edinburgh, 1994

[Ra1] A. Ranta, *Context-relative syntactic categories and the formalization of mathematical text*, S. Berardi and M. Coppo, eds., Types for Proofs and Programs, pp. 231-248, Lecture Notes in Computer Science 1158, Springer-Verlag, Heidelberg, 1996

[Ra2] A. Ranta, *Structures grammaticales dans le français mathématique*, Mathématiques, informatique et Sciences Humaines., vol. 138 pp. 5-56 and 139 pp. 5-36, 1997

[Ra3] A. Ranta, *Grammatical Framework : A Type-Theoretical Grammar Formalism*, in Journal of Functional Programming, , 14(2):145-189, 2004

[RiVo] A. Riazanov et A. Voronkov, *Vampire 1.1 (system description)*, in IJCAR vol 2083, LNAL pp 376-380, 2001

[Ro] M. Roger, *Raffinements de la résolution et vérification de protocoles cryptographiques*, Thèse d'informatique, ENS de Cachan, 24 Octobre 2003

[Ru] P. Rudnicki, *An Overview of the Mizar project*, in B. Nordstrom, K. Petterson and G. Plotkin (eds.) : Proceedings of the 1992 Workshop on Types for Proofs and Programs. Bastad, pp. 311-332, 1992

[Si] D. L. Simon, *Checking natural language proofs*, in E. L. Lusk, R. A. Overbeek (Eds.), 9th International Conference on Automated Deduction, in Lecture Notes in Computer Science, vol. 310, Springer, Berlin, 1988

[SoJo] R. Socher-Ambrosius, P. Johann, *Deduction systems*, Springer-Verlag, 1996

[Sy] D. Syme, *DECLARE : A Prototype Declarative Proof System for Higher Order Logic*, Tech Report, Comp Lab, Univ of Camb, 1997

[Ta1] T. Tammet, *Resolution Theorem Prover for Intuitionistic Logic*, in CADE-13, LNCS 1104, Springer-Verlag, 1996

[Ta2] T. Tammet, *Resolution, inverse method and the sequent calculus*, University of Göteborg and Chalmers University of Technology

[TPS] *TPS Theorem Proving System*, `http://gtps.math.cmu.edu/tps.html`

[We1] M. Wenzel, *Isabelle/Isar - a versatile environment for human-readable formal proof documents*, PhD thesis, Institut für Informatik, Technische Universität München, 2002

[We2] M. Wenzel, *Isar - a Generic Interpretative Approach to Readable Formal Proof Documents*, In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, L. Thery, editors, Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, LNCS 1690, Springer, 1999

[WeWi] M. Wenzel, F. Wiedijk, *A comparison of Mizar and Isar*, 2002

[Za] V. Zammit, *On the Implementation of an Extensible Declarative Proof Language*, in TPHOLs, pp 185-202, 1999

[Zi] C. Zinn, *Supporting the formal verification of mathematical texts*, in Journal of applied Logic 4, pp 592-621, 2006

Liste des notations et des symboles

Notation	Signification	Chapitre
$\mathcal{H} \vdash G$	Séquent en déduction naturelle	2
$\bot$	Formule fausse	2
@	Concaténation de listes	2
$interp_{nc}(nc, G)$	Interprétation d'une commande	2
$interp_{ncs}(ncs, G)$	Interprétation d'une commande simple	2
$interp_{meta}(m, G)$	Interprétation d'une métavariable	2
$\mathcal{I}(H, meta)$	Interprétation simplifiée d'une métavariable	2
$\phi_c(meta)$	Formule dans le cas où un but est changé	2
$\phi_i(meta)$	Formule dans le cas où le but est inchangé	2
$\multimap$	Flèche linéaire	3
$\rightarrow$	Flèche intuitioniste	3
$-?$ ou $-?_i$	Flèche sous-spécifiée	3
$\rightsquigarrow$ ou $\rightsquigarrow_i$	Flèche quelconque	3
λ°	Symbole d'abstraction linéaire	3
λ	Symbole d'abstraction intuitionniste	3
$\lambda^\star$	Symbole d'abstraction quelconque	3
$[\Gamma\ ;\ \Delta] \vdash t : \beta$	Jugement de typage	3
$U(A, B)$	Unificateur le plus général des types A et B	3
$\mathrm{TP}(t)$	Type principal du terme t	3
$f(c, T)$	type rencontré à l'adresse c du type T	3
$h(c, T)$	Tête du type $f(c, T)$	3
$[]$	Adresse vide	3
$c :: d$	Concaténation d'adresses	3
ϵ^k	Adresse avec k ϵ	3
$\varphi(c, E)$	Termes justifiants l'adresse c dans E	3
$\tilde{\varphi}(c, x, E)$	Termes justifiants pour une variable libre	3
$E(C)$	Ensemble associé à la classe C	3
$\mathrm{H}(C)$	Tête de type de la classe C	3
$\Phi(C)$	Ensemble justifiant de la classe C	3
$\mathrm{Cl}(P)$	Classe du point P	3
$\square$	Clause vide	4 et 5
mgu	Unificateur le plus général (most general unifier)	4 et I
$F^\perp$	Négation de la formule F dans une clause	5
LK	Calcul des séquents classique	5
$\Gamma \vdash \Delta$	Séquent dans LK	5
$\vdash_{LK}$	Symbole de dérivabilité dans LK	5
$\psi(C)$	Traduction de clause en formule (sys. logique)	5
$\phi(C)$	Traduction de clause en formule (sys. implémenté)	5

Annexe I

Formules du premier ordre et unification

I.1 Formules du premier ordre

On se donne un ensemble $\mathcal{V}$ de variables (notées x,y,...) et un ensemble $\mathcal{F}$ de fonctions dont l'arité est fixée (notées f,g,...). On peut introduire un ensemble $\mathcal{C}$ de constantes (notées a, b,c,...) ou plutôt considérer que ce sont des fonctions d'arité 0, ce que nous ferons.

Définition I.1.1 (terme) *Un terme t est défini par la grammaire :*

$$t :== \quad x \quad | \quad f(\overrightarrow{t})$$

avec x dans $\mathcal{V}$ et f dans $\mathcal{F}$

On se donne maintenant un ensemble $\mathcal{A}$ de symboles de relations d'arité fixée (notés A,B,...).

Définition I.1.2 (formule atomique) *$Atom = A(t_1, \ldots, t_n)$ est une formule atomique si A est une relation d'arité n ($n \geq 0$) de $\mathcal{A}$ et les t_i sont des termes. On ajoute parfois les constantes $\top$ (vrai) et $\bot$ (faux) dans les atomes. Nous les considérons comme des symboles de relation d'arité 0.*

Définition I.1.3 (formule) *Une formule du premier ordre est définie par la grammaire :*

$$F :== \quad Atom \quad | \quad F \vee F \quad | \quad F \wedge F \quad | \quad F \to F \quad | \quad \neg F \quad | \quad \exists x.F \quad | \quad \forall x.F$$

où x est une variable de $\mathcal{V}$ et $Atom$ est une formule atomique.

Remarque I.1.4 On peut parfois ajouter ou éliminer certains des symboles de relation entre formules. On peut par exemple supprimer $\to$ en considérant que $F_1 \to F_2$ est en fait équivalent à la formule $\neg F_1 \vee F_2$. On peut au contraire ajouter $\leftrightarrow$ comme symbole de relation binaire.

I.2 L'unification

I.2.1 Définitions

Avant de définir l'unification, nous donnons les définitions de la substitution.

Définition I.2.1 (substitution) *Une substitution σ est une application de l'ensemble des variables dans l'ensemble des termes. Nous noterons parfois $v\sigma$ au lieu de $\sigma(v)$.*

Définition I.2.2 (support) *Le support d'une substitution σ est l'ensemble des variables qui ne sont pas laissées fixes par σ. Si le support de σ est fini, $\{x_1, \ldots, x_n\}$ et si pour tout $i \in \{1, \ldots, n\}$ $x_i\sigma = t_i$, alors nous noterons sigma $= \{t_1/x_1, \ldots, t_n/x_n\}$.*

Définition I.2.3 (σ_x) *Soit σ une substitution. Nous notons σ_x la substitution vérifiant :*

$$y\sigma_x = y\sigma \text{ si } y \neq x \text{ et } x\sigma_x = x.$$

Définition I.2.4 (substitution de terme, de formule) *Soit σ une substitution, on définit pour tout terme t sa substitution $t\sigma$ par σ par induction :*
– $v\sigma = v\sigma$ pour v dans $\mathcal{V}$;
– $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$ pour f un symbole de fonction d'arité n.
On définit alors la substitution d'une formule par une substitution σ par induction sur la formule :
– $A(t_1, \ldots, t_n)\sigma = A(t_1\sigma, \ldots, t_n\sigma)$ pour A un symbole de relation d'arité n ;
– $(\neg F)\sigma = \neg(F\sigma)$;
– $(F_1 \circ F_2)\sigma = F_1\sigma \circ F_2\sigma$ pour $\circ$ un symbole de relation binaire entre formules ;
– $(\forall x.F)\sigma = \forall x.(F\sigma_x)$;
– $(\exists x.F)\sigma = \exists x.(F\sigma_x)$.

Remarque I.2.5 Notons au passage que $t\sigma\tau = (t\sigma)\tau = (\tau \circ \sigma)(t)$.

Définition I.2.6 (substitution libre) *Une substitution σ libre est caractérisée par induction :*
– σ est libre pour A si A est atomique ;
– σ est libre pour $\neg F$ si elle est libre pour F ;
– σ est libre pour $F_1 \circ F_2$ si elle l'est pour F_1 et F_2 ;
– σ est libre pour $\forall x.F$ et pour $\exists x.F$ si σ_x est libre pour F et si $y\sigma$ ne contient pas x pour toute variable libre y de F, $y \neq x$.

Pour éviter les problèmes de capture de variables dans l'application d'une substitution σ sur un terme t (resp. une formule F), nous supposerons toujours que σ est libre dans t (resp. F). Pour cela il suffit de renommer éventuellement les variables liées dans t (resp. F) qui apparaissent dans l'image de σ.

Définition I.2.7 (Unificateur) *Soit $E = \{t_1, \ldots, t_n\}$ (resp. $\{F_1, \ldots, F_n\}$) un ensemble de termes (resp. formules), et soit σ une substitution. On dit que σ est un unificateur de E si $t_1\sigma = \cdots = t_n\sigma$ (resp. $F_1\sigma = \cdots = F_n\sigma$). On dit alors que E est unifiable.*

Définition I.2.8 (Unificateur le plus général) *Soient E un ensemble de termes unifiable, et soit σ un unificateur de E. On dit que σ est un unificateur le plus général de E (et on note parfois m.g.u. pour* most general unifier*) si pour tout unificateur θ de E il existe une substitution λ telle que $\theta = \lambda \circ \sigma$.*

I.2.2 Algorithme d'unification

Notre but maintenant va être de donner un algorithme qui cherche, s'il en existe, un unificateur le plus général à un ensemble de termes. Nous n'allons ici donner qu'un algorithme élémentaire. Nous l'appliquerons en général à un ensemble de deux termes dont on cherche à savoir s'ils sont unifiables.
Nous allons tout d'abord donner encore quelques définitions.

Définition I.2.9 (ensemble de désaccord) *Soit W un ensemble de termes non vide. L'ensemble de désaccord de W est obtenu en cherchant la place du premier symbole (en lisant de gauche à droite) à partir de laquelle les termes n'ont plus tous la même écriture, puis en extrayant dans chaque terme le sous-terme qui commence à cette place. L'ensemble de ces sous-termes est l'ensemble de désaccord de W.*

Exemple I.2.10 Soit $W = \{h(x, f(t, g(x), a)), h(x, f(t, y, a)), h(x, f(t, g(x), b))\}$.
L'ensemble de désaccord de W est $\{g(x), y\}$ (seulement deux termes puisque c'est un ensemble et non un n-uplet).

L'algorithme

Entrée : Un ensemble W de termes (distincts) non vide.

Début
$k := 0$
$W_k := W$
$\sigma_k = Id$ (la substitution identité)
Tant que W_k n'est pas un singleton **faire**
$D_k :=$ l'ensemble de désaccord de W_k
Si il existe des éléments v_k et t_k dans D_k tels que v_k est une variable qui n'apparait pas dans t_k
alors $\sigma_{k+1} := \{t_k/v_k\} \circ \sigma_k$
$W_{k+1} = W_k\{t_k/v_k\}$ (notez que $W_{k+1} = W\sigma_{k+1}$)
sinon stop "erreur : W n'est pas unifiable"
fin si
$k := k + 1$
fin tant que
fin

Sortie : σ_k, un unificateur le plus général de W, s'il en existe.

Exemple I.2.11 Si W est celui donné dans l'exemple précédent, alors à la première étape, nous allons obtenir $W_1 = \{P(x, f(t, g(x), a)), P(x, f(t, g(x), b))\}$. Mais par suite $D_1 = \{a, b\}$, et comme a et b ne sont pas des variables mais des constantes distinctes, l'algorithme échoue : W n'est pas unifiable. Par contre les deux premiers éléments de W le sont.

Nous allons maintenant prouver le

Théorème I.2.12 *Soit W un ensemble de termes non vide. L'algorithme termine toujours : Si W est non unifiable, alors l'algorithme le précise. Sinon, il donne l'unificateur le plus général de W.*

Preuve : Le fait que l'algorithme finisse toujours est clair : comme W n'a qu'un nombre fini de variables, l'algorithme ne peut pas faire une suite infinie de W_k.
Si E n'est pas unifiable, supposons que l'algorithme termine sans erreur. Alors la sortie de la boucle "tant que" implique que l'on a trouvé une substitution qui unifie tous les termes. Cela signifie que E est unifiable : Contradiction.
Supposons maintenant que E est unifiable. Nous devons prouver que l'algorithme ne termine pas par une erreur, et que la substitution obtenue est bien l'unificateur le plus général. Pour cela, soit θ un unificateur de W. Nous allons prouver par récurrence que pour tout n (quand σ_n a été défini), il existe une substitution λ_n telle que $\theta = \lambda_n \circ \sigma_n$.
Pour $n = 0$, on pose $\lambda_0 = \theta$. Supposons que $\theta = \lambda_i \circ \sigma_i$ pour $0 \leq i \leq n$.
Si W_n est un singleton, alors σ_n est un unificateur de W et comme $\theta = \lambda_n \circ \sigma_n$, c'est un unificateur le plus général.
Sinon, l'algorithme trouve l'ensemble de désaccord D_n de W_{σ_n}. Comme $\theta = \lambda_n \circ \sigma_n$ unifie W, λ_n unifie D_n. Puisque D_n est un ensemble de désaccord unifiable, il doit y avoir une variable v_n dans D_n. Soit t_n un élément de D_n autre que v_n. Alors $v_n\lambda_n = t_n\lambda_n$. Si v_n apparaît dans t_n, alors $v_n\lambda_n$ apparaît dans $t_n\lambda_n$, ce qui est impossible puisque v_n et t_n sont distincts et que λ_n les unifie. Donc v_n n'apparaît pas dans t_n. Alors l'algorithme ne s'arrête pas pour cause d'erreur à cette étape, mais va choisir un v_n et un t_n tels que v_n n'apparaît pas dans t_n, puis va définir $\sigma_{n+1} := \{t_n/v_n\} \circ \sigma_n$. Voyant les substitutions dans la notation des ensembles, définissons $\lambda_{n+1} = \lambda_n \setminus \{t_n\lambda_n/v_n\}$. Comme v_n n'apparaît pas dans t_n, il est clair que $t_n\lambda_{n+1} = t_n\lambda_n$. Ainsi

$$\begin{aligned}\lambda_{n+1} \circ \{t_n/v_n\} &= \{t_n\lambda_{n+1}/v_n\} \cup \lambda_{n+1} \\ &= \{t_n\lambda_n/v_n\} \cup \lambda_{n+1} \\ &= \{t_n\lambda_n/v_n\} \cup \lambda_n \setminus \{t_n\lambda_n/v_n\} \\ &= \lambda_n\end{aligned}$$

Finalement, on a

$$\theta = \lambda_n \circ \sigma_n = \lambda_{n+1} \circ \{t_n/v_n\} \circ \sigma_n = \lambda_{n+1} \circ \sigma_{n+1}$$

Nous avons donc bien fait la preuve que l'agorithme termine en donnant une substitution, et que cette substitution est un unificateur le plus général de W. □

Remarque I.2.13 [unification simultanée] On peut noter $u \sim v$ l'équation dont la solution est un unificateur s'il existe. Il peut arriver que l'on veuille unifier simultanément des éléments, c'est à dire résoudre un système d'équations $(u_i \sim v_i)_{1\leq i\leq n}$. La solution peut être un unificateur le plus général, de manière analogue à la définition que nous avons déjà vue.

Pour en obtenir un, il suffit d'appliquer l'algorithme suivant, récursif :
Soit σ_n un unificateur le plus général de u_n et v_n.
Si $n = 1$
- alors l'unificateur recherché est σ_1.
- sinon soit σ un unificateur le plus général du système $(u_i\sigma_n \sim v_i\sigma_n)_{1\leq i\leq n-1}$, l'unificateur recherché est $\sigma \circ \sigma_{n+1}$.

L'algorithme termine toujours et il donne une solution si et seulement si le système en a une, et il donne alors l'unificateur le plus général.
La preuve se fait par induction sur le nombre d'équations. Le fait que l'algorithme termine est évident car l'algorithme d'unification termine et le nombre d'équation diminue à chaque appel récursif.
Si il y a une seule équation c'est le théorème que nous venons de prouver.
Supposons que la propriété soit vraie pour n équations et prenons un système à $n+1$ équations.
Soit $u_{n+1} \sim v_{n+1}$ la dernière équation du système. Soit σ_{n+1} un unificateur le plus général de u_{n+1} et v_{n+1}. S'il n'en existe pas, alors u_0 et v_0 ne sont pas unifiables, donc le système ne l'est pas.
Sinon appliquons σ_{n+1} à l'ensemble E des autres équations du système. Si $E\sigma_{n+1}$ a une solution σ, prouvons que $\sigma \circ \sigma_{n+1}$ est un unificateur le plus général. Il est clair que c'est un unificateur.
Soit τ un unificateur du système. L'équation $u_{n+1} \sim v_{n+1}$ a pour unificateur le plus général σ_{n+1}. Donc il existe μ telle que $\tau = \mu \circ \sigma_{n+1}$. Par conséquent μ est un unificateur de $E\sigma_{n+1}$ et comme σ en est un unificateur le plus général μ s'écrit $\mu' \circ \sigma$. Ainsi $\tau = \mu' \circ \sigma \circ \sigma_{n+1}$: $\sigma \circ \sigma_{n+1}$ est un unificateur le plus général.
Si $E\sigma_{n+1}$ n'a pas de solution alors le système n'a pas de solution car sinon en prenant τ un unificateur on sait que $\tau = \mu \circ \sigma_{n+1}$ par ce que nous avons vu plus haut, donc $E\sigma_{n+1}$ a une solution : μ. □

Remarque I.2.14 Nous avons donné un algorithme d'unification de termes, mais nous n'avons pas parlé d'unification de formules. Deux formules sont égales si elles sont identiques à renomage des variables liées près. Il est possible d'unifier deux formules grâce à l'unification sur les termes.

En effet, voici l'algorithme qui généralement résout un ensemble d'équations entre formules :

Entrée : Un ensemble W d'équations entre formules non vide.

Début
$k := 0$

$W_k := W$
$\sigma_k = Id$ (la substitution identité)
Tant que W_k n'est pas un singleton **faire**
Soit $F \sim G$ une équation de $W_k = \{F \sim G\} \cup W'_k$.
cas :
– $F = A(t_1, \ldots, t_n)$ et $G = A(u_1, \ldots, u_n)$
-> Soit σ l'unificateur le plus général simultané de $\{t_i = u_i\}_{1 \leq i \leq n}$.
$W_{k+1} := W'_k\sigma$
$\sigma_{k+1} := \sigma \circ \sigma_k$
– $F = F_1 \wedge F_2$ et $G = G_1 \wedge G_2$
->
$W_{k+1} := W'_k \cup \{F_1 = G_1, F_2 = G_2\}$
$\sigma_{k+1} := \sigma_k$
– $F = F_1 \vee F_2$ et $G = G_1 \vee G_2$
->
$W_{k+1} := W'_k \cup \{F_1 = G_1, F_2 = G_2\}$
$\sigma_{k+1} := \sigma_k$
– $F = F_1 \to F_2$ et $G = G_1 \to G_2$
->
$W_{k+1} := W'_k \cup \{F_1 = G_1, F_2 = G_2\}$
$\sigma_{k+1} := \sigma_k$
– $F = \forall x.F'(x)$ et $G = \forall y.G'(y)$
->
Soit f un nouveau symbole de fonction.
$W_{k+1} := W'_k \cup \{F'(f(x_i)) = G'(f(x_i))\}$ où les x_i sont les variables libres de W_k
$\sigma_{k+1} := \sigma_k$
– $F = \exists x.F'(x)$ et $G = \exists y.G'(y)$
->
Soit f un nouveau symbole de fonction.
$W_{k+1} := W'_k \cup \{F'(f(x_i)) = G'(f(x_i))\}$ où les x_i sont les variables libres de W_k
$\sigma_{k+1} := \sigma_k$
– Autre cas
->
stop "erreur : W n'est pas unifiable"
fin cas
$k := k + 1$
fin tant que
fin

Sortie : σ_k, un unificateur le plus général de W, s'il en existe.

Théorème I.2.15 *Soit W un ensemble de formules non vide. L'algorithme termine toujours : Si W est non unifiable, alors l'algorithme le précise. Sinon, il donne l'unificateur le plus général de W, dont le domaine est inclus dans les variables libres du système.*

Preuve : Il se fait par induction sur la somme des complexités des formules de W. Regardons les cas selon l'équation $F \sim G$ de $W = \{F \sim G\} \cup W'$ qui est prise :
– $F = A(t_1, \ldots, t_n)$ et $G = A(u_1, \ldots, u_n)$: F et G sont unifiables si et seulement si le système $\{t_i \sim u_i\}$ a une solution. L'algorithme d'unification donne l'unificateur le plus général du système. Une preuve semblable à la preuve de l'unification de système nous donne alors le résultat.
– $F = F_1 \wedge F_2$ et $G = G_1 \wedge G_2$: F et G sont unifiables si et seulement si le système $\{F_1 \sim G_1, F_2 \sim G_2\}$ a une solution. L'induction permet de terminer.
– $F = F_1 \vee F_2$ et $G = G_1 \vee G_2$: F et G sont unifiables si et seulement si le système $\{F_1 \sim G_1, F_2 \sim G_2\}$ a une solution. L'induction permet de terminer.

- $F = F_1 \to F_2$ et $G = G_1 \to G_2$: F et G sont unifiables si et seulement si le système $\{F_1 \sim G_1, F_2 \sim G_2\}$ a une solution. L'induction permet de terminer.
- $F = \forall x.F'(x)$ et $G = \forall y.G'(y)$: x pour F et y pour G étant des variables liées, elles ne peuvent être utilisées pour unifier le système. Donc si $F \sim G$ a une solution σ, alors $F'(f(x_i)) \sim G'(f(x_i))$ a aussi pour solution σ.
  Si $F'(f(x_i)) \sim G'(f(x_i))$ a une solution σ, alors f ne peut pas appartenir à l'image de σ, car cela signifierait qu'à une étape de l'unification on a substitué à une variable libre un terme contenant f. Cela est impossible car f a pour variables libres toutes les variables libres du système et par conséquent la condition qu'une variable ne doit pas appartenir au terme par lequel elle est substituée ne serait pas vérifiée. Ainsi $F'(f(x_i))\sigma = G'(f(x_i))\sigma$ implique que $F\sigma = G\sigma$.
  Nous avons alors l'équivalence, et l'induction permet de terminer.
- analogue au cas précédent.
- Autre cas : dans tous les autres cas les formules ne sont pas unifiables. □

Annexe II

Contradiction séquentielle

Le but de cette annexe est d'expliquer le moyen mis en oeuvre pour gérer les clauses de séparation vues au chapitre 4 à la section 4.3.2. Plus particulièrement nous expliquons comment l'ajout d'une nouvelle clause de séparation à un ensemble déjà existant est traité.
Nous sommes ici dans un cas propositionnel, puisque rappelons-le les clauses de séparation sont des clauses contenant des littéraux propositionnels (atomes propositionnels ou négation d'atomes propositionnels). Le but est de construire et de mettre à jour un arbre binaire qui rend compte des interprétations possibles des littéraux.
En effet, le démonstrateur construit les clauses de séparation au fur et à mesure, pendant la preuve. Au lieu de chercher immédiatement une preuve par résolution ces clauses, il est plus rapide de tester si l'ajout d'une nouvelle clause rend l'ensemble contradictoire de la manière que nous allons donner.
Ensuite seulement nous cherchons une preuve par résolution, par une méthode détaillée dans le chapitre 4, la résolution linéaire, qui est une méthode efficace de résolution utilisant le fait que la dernière clause ajoutée rend l'ensemble contradictoire.

II.1 Définitions

Les définitions données ici sont locales à cette annexe. Nous redéfinissons en effet les clauses d'une manière particulière qui n'est pas valable dans tout le reste.

Définition II.1.1 (littéral, clause) *Soit $\mathcal{P}$ un ensemble au moins dénombrable d'élements notés p, q, a,. . . .*
- *$\mathcal{P}$ est appelé l'ensemble des variables propositionnelles. Ainsi une variable propositionnelle est un élément de $\mathcal{P}$.*
- *Un littéral est une variable propositionnelle (a) ou la négation d'une variable propositionnelle ($\neg a$).*
- *Une clause est une liste (ordonnée) de littéraux, sans répétition et sans tautologie (c'est à dire qu'il ne peut y avoir à la fois a et $\neg a$ dans une clause).*

Définition II.1.2 *On dit qu'une variable propositionnelle apparaît dans une clause si elle ou sa négation appartient à la clause. On dit alors que la variable est respectivement positive et négative dans la clause.*

Un ensemble de clauses n'existe pas ici. Nous parlerons de listes (ordonnées) de clauses. Nous noterons tout de même S une telle liste. Si l est un littéral et C est une clause, nous noterons $l \cdot C$ la clause ayant l comme premier élément et C comme queue. Si S est une liste de clauses nous noterons $S \cdot C$ la liste avec les éléments de S en premier et C en dernier.

Définition II.1.3 (arbre d'interprétations) *Un arbre d'interprétations t est défini par la grammaire suivante :*

$$t \quad ::= \quad \top \quad | \quad \bot \quad | \quad n(a, t, t)$$

où a est dans $\mathcal{P}$

L'idée d'un arbre d'interprétations est que les feuilles donnent l'interprétation de la suite de clauses en fonction du chemin suivi depuis la racine de l'arbre. Aller à gauche (resp. droite) d'un noeud signifie prendre l'interprétation dans laquelle la variable propositionnelle est vraie (resp. fausse). La feuille $\top$ (resp. $\bot$) signifie que la liste de clauses est satisfiable (resp. insatisfiable).

Définition II.1.4 *Un arbre d'interprétations est dit clos si toutes ses feuilles sont $\bot$, autrement dit s'il ne contient pas de $\top$.*

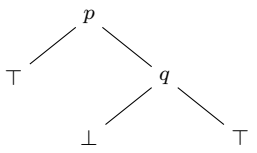
Nous parlerons dans la suite plus simplement d'arbre au lieu d'arbre d'interprétations. Nous n'avons pas défini l'insatisfiabilité d'une liste de clauses, nous utilisons pour ceci la définition usuelle pour les ensembles de clauses (dans le cas de la logique propositionnelle).

Définition II.1.5 (arbre associé à une clause) *L'arbre associé à une clause est défini par induction sur la taille de la clause :*
- $t_0([]) = \bot$;
- $t_0(a \cdot C') = n(a, \top, t_0(C'))$;
- $t_0(\neg a \cdot C') = n(a, t_0(C'), \top)$.

Exemple II.1.6 Soit $C_1 = p \cdot \neg q$. Alors

$$t_0(C_1) = n(p, \top, t_0(\neg q)) = n(p, \top, n(q, t_0([]), \top)) = n(p, \top, n(q, \bot, \top))$$

Nous pouvons le représenter ainsi :



Il n'y a qu'une interprétation dans laquelle la clause est fausse : celle où p est fausse et q est vraie.

Définition II.1.7 (mise à jour d'un arbre) *La mise à jour d'un arbre t par une clause C est définie par cet algorithme :*

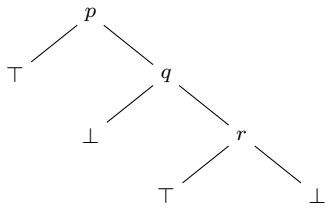
$add(C, t) =$
Si $C = []$ *Alors* $\bot$
Sinon regardons les cas pour t *:*
  Si $t = \bot$ *Alors* $\bot$
  Si $t = \top$ *Alors* $t_0(C)$
  Si $t = n(a, t_1, t_2)$ *Alors*
    Si la variable propositionnelle a apparait dans C
    Alors notons C' la clause C sans la variable a.
      Si a est positive dans C Alors $n(a, t_1, add(C', t_2))$
      Sinon $n(a, add(C', t_1), t_2)$
    Sinon $n(a, add(C, t_1), add(C, t_2))$

La terminaison de l'algorithme est claire puisqu'il y a toujours dans les cas non terminaux au moins un des arguments qui est plus petit et aucun n'est plus grand.

Exemple II.1.8 Soit $t = t_0(C_1)$, où C_1 est la clause vue à l'exemple II.1.6. Soit la clause $C_2 = r \cdot p \cdot q$. Alors

$$\begin{aligned} add(C_2, t) &= add(r \cdot p \cdot q, n(p, \top, n(q, \bot, \top))) \\ &= n(p, \top, add(r \cdot q, n(q, \bot, \top))) \\ &= n(p, \top, n(q, \bot, add(r, \top))) \\ &= n(p, \top, n(q, \bot, t_0(r))) \\ &= n(p, \top, n(q, \bot, n(r, \top, t_0([])))) \\ &= n(p, \top, n(q, \bot, n(r, \top, \bot))) \end{aligned}$$

Ce que l'on peut représenter ainsi :



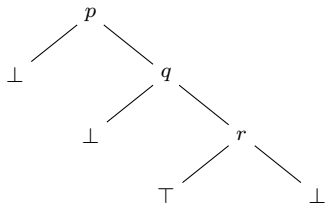
Le littéral r a été rajouté seulement au bout de la branche représentant l'interprétation qui rend p et q fausses car dans les autres cas la clause C_2 est valide.

Définition II.1.9 (arbre associé à une liste de clauses)
L'arbre associé à une liste de clauses est défini par induction sur la taille de la liste :
– $arbre([]) = \top$;
– $arbre(S \cdot C) = add(C, arbre(S))$.

Exemple II.1.10 Soit $S = C_1 \cdot C_2 \cdot C_3$ où C_1 et C_2 sont les clauses définies dans les exemples II.1.6 et II.1.8, et C_3 est la clause $\neg p$. Rappelons que nous avons déjà donné $t = t_0(C_1)$, ainsi que $add(C_2, t)$. Alors :

$$\begin{aligned} arbre(S) &= add(C_3, add(C_2, add(C_1, []))) \\ &= add(\neg p, add(C_2, add(C_1, \top))) \\ &= add(\neg p, add(C_2, t_0(C_1))) \\ &= add(\neg p, add(C_2, t)) \\ &= add(\neg p, n(p, \top, n(q, \bot, n(r, \top, \bot)))) \\ &= n(p, \bot, n(q, \bot, n(r, \top, \bot))) \end{aligned}$$

Ce que l'on peut interpréter ainsi :



Une seule interprétation des variables p, q et r rend l'ensemble S satisfiable. Il s'agit de celle qui rend p et q fausses et r vraie.

Définition II.1.11 *Soit S une liste de clauses et a une variable propositionnelle. Nous notons S_a (resp. $S_{\neg a}$) la liste de clauses définie comme étant S dans laquelle on a enlevé les clauses contenant le littéral a (resp $\neg a$) et dans laquelle on a enlevé le littéral $\neg a$ (resp. a) dans les clauses restantes.*

II.2 Premiers lemmes

Lemme II.2.1 *Soit S une liste de clauses non vide. Alors $arbre(S) \neq \top$.*

Preuve : Par récurrence sur la taille de S. Si S n'a qu'un élément C, alors $arbre(S) = add(C, \top) = t_0(C)$ d'après l'algorithme. Si $C = []$ alors c'est $\bot$ sinon il s'agit d'un arbre commençant par un noeud. Pour l'étape d'induction nous supposons que $S = S' \cdot C$ avec S' non vide. Alors $arbre(S) = add(C, arbre(S'))$. Par hypothèse de récurrence $arbre(S')$ est différent de $\top$, c'est donc soit $\top$, et le raisonnement du cas de base nous permet de conclure, soit un noeud $n(a, t_1, t_2)$. Dans ce cas une étude de l'algorithme nous montre que $arbre(S)$ est un noeud $n(a, t'_1, t'_2)$. □

Lemme II.2.2 *Soit S une liste de clauses. Alors $arbre(S) = \top$ si et seulement si $S = []$.*

Preuve : Ceci est clair grâce au lemme précédent pour le sens gauche droite par contraposée, et par définition pour le sens droite gauche. □

Lemme II.2.3 *Soit t un arbre clos et C une clause. Alors $add(C, t)$ est clos.*

Preuve : Par induction sur la complexité de t. Si la clause C est vide, c'est clair puisqu'alors $add(C, t) = \bot$. Si $t = \bot$, c'est la même chose. Sinon on a $t = n(a, t_1, t_2)$ avec t_1 et t_2 clos. On voit alors en étudiant l'algorithme que dans tous les cas $add(C, t) = n(a, t'_1, t'_2)$ avec t'_1 et t'_2 clos (s'ils sont de la forme $add(C', t_i)$ ils le sont par induction). □

Lemme II.2.4 *Soit S une liste (non vide) de clauses. Supposons que $arbre(S) = n(a, t_1, t_2)$. Alors t_1 (resp. t_2) est l'arbre de S_a (resp. $S_{\neg a}$).*

Preuve : Elle se fait par induction sur le couple (nombre de variables propositionnelles dans S, taille de S). Comme $arbre(S) = n(a, t_1, t_2)$, S n'est pas vide. Si $S = [C]$, alors $arbre(S) = t_0(C)$ et soit $C = a \cdot C'$ soit $C = \neg a \cdot C'$. Dans le premier cas $t_0(a \cdot C') = n(a, \top, t_0(C'))$. Or nous avons $S_a = []$ et $S_{\neg a} = [C']$, d'où le résultat. Dans le second cas $t_0(\neg a \cdot C') = n(a, t_0(C'), \top)$. Or $S_a = [C']$ et $S_{\neg a} = []$, d'où le résultat.
Si $S = S' \cdot C$ avec S' non vide, alors $arbre(S) = add(C, arbre(S'))$.
Comme $arbre(S) = n(a, t_1, t_2)$, il est impossible que $arbre(S') = \bot$ et comme $S' \neq []$ il est aussi impossible que $arbre(S') = \top$ par le lemme II.2.3. Nécessairement, $arbre(S') = n(a, t'_1, t'_2)$ (la variable propositionnelle doit être a). Par induction, t'_1 est l'arbre de S'_a et t'_2 est l'arbre de $S'_{\neg a}$. Nécessairement C est non vide, car sinon $arbre(S) = \bot$. Regardons les cas selon l'algorithme :

- Si a apparaît dans C et est positif alors $S_a = S'_a$ et $S_{\neg a} = S'_{\neg a} \cdot C'$ avec C' la clause C sans a. Or $arbre(S) = n(a, t'_1, add(C', t'_2))$ donc $t_1 = t'_1$ est l'arbre de $S'_a = S_a$ et $add(C', t'_2)$ est l'arbre de $S'_{\neg a} \cdot C' = S_{\neg a}$.
- Si a apparaît dans C et est négatif alors un raisonnement analogue au précédent cas permet de conclure.
- Si a n'apparaît pas dans C alors $S_a = S'_a \cdot C$ et $S_{\neg a} = S'_{\neg a} \cdot C$. Or $arbre(S) = n(a, add(C, t'_1), add(C, t'_2))$ et comme $add(C, t'_1) = arbre(S'_a \cdot C) = arbre(S_a)$ et $add(C, t'_2) = arbre(S'_{\neg a} \cdot C) = arbre(S_{\neg a})$ nous avons bien le résultat cherché. □

II.3 La proposition

Proposition II.3.1 *Soit S une suite de clauses. Alors S est insatisfiable si et seulement si $arbre(S)$ est clos.*

Preuve : Elle se fait par induction sur le couple (nombre de variables propositionnelles dans S, taille de S). Si $S = []$ alors S n'est pas insatisfiable et $arbre(S) = \top$ n'est pas clos. Si $S = S' \cdot C$ alors $arbre(S) = add(C, arbre(S'))$.

- Supposons S insatisfiable. Si S' est insatisfiable alors par induction $arbre(S')$ est clos et $add(C, arbre(S'))$ est clos également par le lemme II.2.3. Sinon $arbre(S')$ n'est pas clos. Si $C = []$ alors c'est terminé car $arbre(S) = \bot$. Sinon nous regardons les cas selon $arbre(S')$:
  - $\bot$: impossible car il est non clos.
  - $\top$: alors S' est vide par le lemme II.2.2. Mais comme C est non vide, $S = \{C\}$ ne peut pas être insatisfiable, donc c'est impossible.
  - $n(a, t_1, t_2)$: par le lemme II.2.4 t_1 est l'arbre de S'_a et t_2 est l'arbre de $S'_{\neg a}$. Comme S est insatisfiable, S_a et $S_{\neg a}$ sont tous deux insatisfiables, et leurs arbres sont clos par induction (a apparaît bien dans les clauses de S, donc S_a et $S_{\neg a}$ ont moins de variables propositionnelles). Suivons l'algorithme :
    - Si a apparaît dans C et est positif alors $S_a = S'_a$ et $S_{\neg a} = S'_{\neg a} \cdot C'$ avec C' la clause C sans a. Or $arbre(S) = n(a, t_1, add(C', t_2))$ et comme $t_1 = arbre(S'_a) = arbre(S_a)$ est clos et $add(C', t_2) = arbre(S'_{\neg a} \cdot C') = arbre(S_{\neg a})$ est clos également. Donc l'arbre est clos.
    - Si a apparaît dans C et est négatif alors un raisonnement analogue au précédent cas permet de conclure que $arbre(S)$ est clos.
    - Si a n'apparaît pas dans C alors $S_a = S'_a \cdot C$ et $S_{\neg a} = S'_{\neg a} \cdot C$. Or
      $arbre(S) = n(a, add(C, t_1), add(C, t_2))$ et comme
      $add(C, t_1) = arbre(S'_a \cdot C) = arbre(S_a)$ est clos et
      $add(C, t_2) = arbre(S'_{\neg a} \cdot C) = arbre(S_{\neg a})$ est clos nous pouvons conclure que $arbre(S)$ est clos.
- Supposons réciproquement S satisfiable. Alors cela implique que soit S_a est satisfiable, soit $S_{\neg a}$ est satisfiable. Or $arbre(S) = add(C, arbre(S'))$. Nécessairement C est non vide car sinon S serait insatisfiable. Comme S est satisfiable, S' l'est aussi et par induction $arbre(S')$ est non clos donc différent de $\bot$ en particulier. Si $arbre(S') = \top$ alors $arbre(S) = t_0(C)$ qui est non clos car C est non vide. Si $arbre(S') = n(a, t_1, t_2)$ alors $arbre(S) = n(a, t'_1, t'_2)$, avec par le lemme II.2.4 t'_1 l'arbre de S_a et t'_2 l'arbre de $S_{\neg a}$. L'induction permet alors de conclure que soit t'_1 soit t'_2 est non clos (le nombre de variables propositionnel a en effet diminué). □

II.4 Simplification

Nous décrivons maintenant une méthode permettant de réduire la taille des arbres pour pouvoir tester la contradiction de ceux-ci.

Définition II.4.1 (simplification d'un arbre) *La simplification d'un arbre est défini par l'algorithme suivant :*

$simpl(t)$ = *Selon les cas sur* t :
  Si $t = \bot$ *Alors* $\bot$
  Si $t = \top$ *Alors* $\top$
  Si $t = n(a, t_1, t_2)$ *Alors*
    Soit $t'_1 = simpl(t_1)$
    Soit $t'_2 = simpl(t_2)$
    Si $t'_1 = t'_2 = \bot$ *alors* $\bot$
    Sinon $n(a, t'_1, t'_2)$

La terminaison de cet algorithme est claire.

Lemme II.4.2 *Soit t un arbre. Alors t est clos si et seulement si $simpl(t) = \bot$.*

Preuve : Par induction sur la complexité de t, si $t = \top$ ou $\bot$ alors $simpl(t) = t$ et le résultat est clair. Si $t = n(a, t_1, t_2)$ alors t est contradictoire si et seulement si t_1 et t_2 sont contradictoire, ce qui est équivalent par induction au fait que $simpl(t_1) = \bot$ et $simpl(t_2) = \bot$, ce qui donne $simpl(t) = \bot$. □

Il y a deux manières de construire de manière séquentielle un arbre d'interprétations. Il est possible de mettre à jour l'arbre simplement en ajoutant les nouvelles clauses. Une fois l'ajout d'une clause effectué on teste si la simplification donne l'arbre $\bot$. On peut également simplifier après chaque ajout l'arbre obtenu, et ajouter les clauses à l'arbre simplifié. La seconde méthode est *a priori* légèrement plus efficace, puisque par définition les arbres sur lesquels nous ajoutons des clauses sont simplifiés par rapport à ceux de la première méthode. Si la première méthode ne pose pas de problème pour le but recherché grâce à la proposition de la section précédente et le lemme ci-dessus, nous devons prouver que les deux méthodes sont équivalentes.

Définition II.4.3 *$simpl^1$ est défini par $simpl^1(S) = simpl(arbre(S))$.*
$simpl^*$ est défini par induction sur la taille de S :
$simpl^*([]) = \top$ et $simpl^*(S' \cdot C) = simpl(add(C, simpl^*(S')))$.

Lemme II.4.4 *Soit t un arbre. Alors $simpl(simpl(t)) = simpl(t)$*

Preuve : Par induction sur la complexité de t. Si $t = \bot$ alors dans les deux cas on obtient $\bot$. Si $t = \top$ alors dans les deux cas on obtient $\top$. Si $t = n(a, t_1, t_2)$ alors deux cas :
- $simpl(t_1) = simpl(t_2) = \bot$: alors $simpl(t) = \bot$ et $simpl(simpl(t)) = \bot$.
- $simpl(t_1)$ ou $simpl(t_2)$ est différent de $\bot$:
  alors $simpl(t) = n(a, simpl(t_1), simpl(t_2))$.
  Or par induction $simpl(simpl(t_i)) = simpl(t_i)$ pour $i = 1, 2$, donc comme l'un des deux est différent de $\bot$ on a bien $simpl(simpl(t)) = simpl(t)$. □

Lemme II.4.5 *Soit C une clause et S une liste de clauses. Alors*

$$simpl(add(C, arbre(S))) = simpl(add(C, simpl(arbre(S))))$$

Preuve : Par induction sur la structure de $arbre(S)$. Si C est la clause vide, alors les deux arbres sont $\bot$, donc nous pouvons supposer C non vide. Regardons selon la forme de $arbre(S)$ les arbres obtenus pour le terme de gauche et le terme de droite de l'égalité :
- $\top$: dans les deux cas l'arbre est $simpl(t_0(C))$ (qui est en fait $t_0(C)$, mais que nous n'avons pas besoin de prouver).
- $\bot$: dans les deux cas l'arbre est $\bot$.
- $n(a, t_1, t_2)$: regardons le cas où $simpl(t_1) = \bot$ et $simpl(t_2) = \bot$. Alors à droite nous obtenons $\bot$. A gauche, $arbre(S)$ est clos, et l'on sait par le lemme II.2.3 que $add(C, arbre(S))$ est également clos. Donc par le lemme II.4.2 on a également $\bot$.
  Dans le cas ou soit $u_1 := simpl(t_1)$ soit $u_2 := simpl(t_2)$ est différent de $\bot$, nous avons $simpl(arbre(S)) = n(a, u_1, u_2)$. Par le lemme II.2.4 nous savons que t_1 est l'arbre de S_a et t_2 est l'arbre de $S_{\neg a}$
  Regardons les cas selon C (non vide) :
  - a apparaît dans C et est positive :
    alors $add(C, arbre(S)) = n(a, t_1, add(C', t_2))$
    et $add(C, simpl(arbre(S))) = n(a, u_1, add(C', u_2))$ avec C' la clause C sans a.
    Or $simpl(add(C', t_2)) = simpl(add(C', u_2))$ par induction, et comme $u_1 = simpl(u_1) = simpl(t_1)$ (la première égalité par le lemme II.4.4) nous obtenons bien l'égalité recherchée.
  - a apparaît dans C et est négative : la preuve est analogue au cas précédent.
  - a n'apparaît pas dans C :
    alors $add(C, arbre(S)) = n(a, add(C, t_1), add(C, t_2))$
    et $add(C, simpl(arbre(S))) = n(a, add(C, u_1), add(C, u_2))$.
    Or par induction $simpl(add(C, t_i)) = simpl(add(C, u_i))$, donc nous obtenons bien l'égalité recherchée. □

Proposition II.4.6 *Soit S une liste de clauses. Alors*

$$simpl^1(S) = simpl^*(S)$$

Preuve : Par induction sur la taille de S. Si $S = []$ alors dans les deux cas nous obtenons $\top$. Si $S = S' \cdot C$, alors
$simpl^1(S) = simpl(add(C, arbre(S')))$ et $simpl^*(S) = simpl(add(C, simpl^*(S')))$. Or par induction $simpl^*(S') = simpl^1(S') = simpl(arbre(S'))$. Or par le lemme précédent
$simpl(add(C, arbre(S'))) = simpl(add(C, simpl(arbre(S'))))$ □

Annexe III

Structure du démonstrateur

Le but de cette annexe n'est certainement pas de donner le code complet du démonstrateur, mais d'en donner la structure générale. Précisons que nous avons programmé ce démonstrateur avec OCaml, qui est un langage de programmation basé sur le langage ML, utilisé pour développer PhoX, Isabelle, HOL entre autres.

III.1 Les divers modules

Le démonstrateur a été découpé en plusieurs modules afin d'alléger l'implémentation. Nous allons définir les modules créés et décrire leur utilité. Nous les trions par ordre de dépendance, le dernier étant le démonstrateur lui-même.

- typespoids : défini le type des poids, et les fonctions permettant de calculer le poids et l'indice des clauses après résolution, décomposition ou contraction. Le poids est la valeur selon laquelle les clauses sont triées. Une clause qui a un poids faible sera utilisée avant une clause ayant un poids élevé. L'indice est une notion de nombre d'utilisations, c'est à dire que l'on indique si une clause peut ou non être beaucoup utilisée pour faire la preuve. Il se peut par exemple qu'une hypothèse doive être utilisée, donc il lui est donné un poids faible, mais une seule fois. On donnera alors un indice élevé à cette hypothèse.
- splitting : module qui s'occupe de la gestion des clauses de séparation (ne contenant que des littéraux de séparation), en mettant à jour un arbre sémantique. Les fonctions de mise à jour détectent les moments où la clause ajoutée rend l'ensemble contradictoire (cf. annexe II pour plus de précisions).
- ptypes : défini les types utilisés par le démonstrateur : les clauses, les ensembles de clauses. Il donne les valeurs par défaut des paramètres et définit les exceptions principales.
- oldeduction : module qui s'occupe des preuves par résolution sur les clauses de séparation, en utilisant la méthode de résolution linéaire ordonnée. Il suffit de donner un ensemble satisfaisable et une clause rendant cet ensemble contradictoire, et une preuve par résolution est retournée (cf. chapitre 4).
- affichage : permet de faire les affichages des clauses et de la preuve.
- majlistes : fonction qui met à jour les ensembles de clauses. Il y a deux ensembles de clauses : les clauses candidates et les clauses utilisées. On gère dans ce module la subsomption et les tautologies.
- prover.ml : module principal dans lequel est définie la fonction *prove* qui effectue la preuve par résolution avec décomposition d'une formule étant données éventuellement des hypothèses.

III.2 Les types de données

- Une clause est un enregistrement contenant les éléments suivants :
  - l'ensemble des littéraux positifs ;
  - l'ensemble des littéraux négatifs ;
  - l'ensemble des littéraux qui sont des méta-variables ;

  - le poids ;
  - l'indice ;
  - l'historique ;
  - les contraintes.
- Un littéral est lui aussi un enregistrement, contenant les éléments suivants :
  - la formule ;
  - le numéro de la formule dont il est originel ;
  - l'historique des règles utilisées pour aboutir à cette formule.
- Une clause décomposable est un enregistrement contenant les éléments suivants :
  - une formule (principale) ;
  - une clause (la clause sans la formule principale) ;
  - un booléen indiquant la positivité de la formule principale dans la clause ;
  - le numéro de la clause dont la clause est originelle.

  On définit alors une liste des clauses décomposables ;
- Les clauses candidates se trouvent dans une liste de clauses triée par ordre croissant de poids.
- L'ensemble des clauses utilisées se trouve dans une liste de couples (clause,table de hashage). Les éléments de la table de hashage sont des clauses avec une formule principale qui sont les décomposées de la clause associée à la table de hashage. La clé de la table est une formule de tête et un booléen indiquant la positivité de la formule principale. Cela permet de réduire les recherches lors de l'application de la résolution, en ne prenant que des clauses se ressemblant le plus possible. Seule la formule principale peut subir la résolution, les autres formules devant attendre la résolution sur la formule principale pour pouvoir devenir principales.
- L'ensemble de toutes les clauses créées est conservé afin de permettre l'affichage de la preuve. En effet, certaines stratégies permettant d'éliminer des clauses, il faut tout de même toutes les conserver dans une autre liste afin de pouvoir les retrouver.

III.3 Le démonstrateur

- Au démonstrateur sont donnés un ensemble de formules (qui est un ensemble d'hypothèses) et une formule (qui est la formule à prouver). Il est possible de donner aux hypothèses des poids et des indices.
- Lors de l'appel du démonstrateur on peut optionnellement préciser un certain nombre de valeurs :
  - verbose : le mode verbeux, qui est un entier. Plus l'entier est grand, plus l'affichage est important ;
  - pause : un booléen qui autorise ou non l'arrêt pendant la preuve à certaines étapes ou non ;
  - maxcndts : le nombre maximum de candidats que le démonstrateur peut conserver ;
  - splitting : un booléen qui autorise ou non la séparation des clauses ;
  - check_subcase : un booléen qui autorise ou non la vérification des sous-cas pour le cas où l'on sépare les clauses. Si le booléen est positif, on ne fait des résolutions que si l'on est dans un même sous-cas ;
  - trace_proof : un booléen qui autorise ou non la mise en mémoire de la preuve, afin de l'afficher à la fin. Il ne s'agit en aucun cas d'une vérification algorithmique de la preuve. La vérification d'une preuve est donc laissée au lecteur qui peut lire la preuve affichée ;
  - timemax : le temps maximum donné au démonstrateur. Le changement de cette variable ne change pas la preuve, aucune notion de temps ayant été introduite dans le démonstrateur pour l'instant (comme c'est le cas dans le démonstrateur vampire par exemple) ;
  - lazymode : un booléen qui interdit ou non de décomposer beaucoup les formules dans les clauses. Cela permet dans les preuves où il faut très peu décomposer de gagner du temps. Dans l'autre cas la décomposition n'est tout de même pas maximale, car seules

les résolutions sur des formules décomposées est testée, mais les décompositions peuvent alors faire boucler le programme dans le cas d'une logique d'ordre supérieur où les décompositions peuvent être infinies ;
– methode : donne la méthode de résolution, qui peut être binaire quelconque, positive ou négative.

- La première étape est de transformer les hypothèses et le but en clause. Pour cela on teste la positivité des formules (après avoir éliminé toutes les doubles négations), puis on élimine toutes les négations et on forme une clause en mettant la formule dans le bon ensemble. La formule à prouver subit auparavant une négation supplémentaire. Chacune des nouvelles clauses créées rentre dans l'ensemble des clauses candidates.
- Ensuite le schéma est général. Indiquons tout d'abord la méthode appliquée originellement : On enlève de la liste des clauses candidates la clause la plus légère. On décompose cette clause de manière "minimale", c'est à dire que l'on décompose chaque littéral de la clause sans décomposer les autres en itérant cette action sur les décomposées des littéraux.

Exemple III.3.1 Par exemple, si la clause est $A \to B, C \wedge (D \vee E)$, alors on décompose $A \to B$, ce qui donne $\neg A, B, C \wedge (D \vee E)$. Puis on décompose la seconde formule, ce qui donne $A \to B, C$ et $A \to B, D \vee E$. La décomposition de cette formule donne encore une formule décomposable $(D \vee E)$, on la décompose donc, ce qui donne $A \to B, D, E$. Au final, voici l'ensemble des décomposées de $A \to B, C \wedge (D \vee E)$:
– $\neg A, B, C \wedge (D \vee E)$;
– $A \to B, C$;
– $A \to B, D \vee E$;
– $A \to B, D, E$.
En fait comme nous l'avons vu dans la section III.2, les décomposées d'une clause sont des clauses avec une formule principale. Ainsi la clause $\neg A, B, C \wedge (D \vee E)$ est représentée deux fois, car il y a deux formules principales, $\neg A$ et B. Remarquons donc que toutes les décompositions ne sont pas effectuées. Par exemple B, D, E n'est pas présente dans les décompositions.

- Une fois les décompositions effectuées on cherche à appliquer la résolution des décomposées avec les autres décomposées, de la même clause et des clauses utilisées. C'est à dire que l'on cherche pour chaque formule principale toutes les autres formules principales qui ont le signe opposé et la même formule de tête, et on tente l'unification.
- Si deux formules sont unifiables, alors on applique la règle de résolution, en cherchant en même temps à appliquer la contraction, c'est à dire que l'on applique la règle suivante :

$$\frac{\Gamma, A_1, \ldots, A_n \quad \Delta, {A'_1}^{\perp}, \ldots, {A'_m}^{\perp}}{\Gamma\sigma, \Delta\sigma} \text{ où } \sigma = \mathrm{mgu}(A_1, \ldots, A_n, A'_1, \ldots, A'_m)$$

En effet nous savons que la contraction est une règle nécessaire pour avoir la complétude du système, mais celle-ci peut être intégrée à la règle de résolution, ceci afin de tester le moins souvent possible cette règle.
- La clause obtenue par résolution (ainsi que chaque éventuelle contractée) est ensuite ajoutée aux clauses candidates. Une mise à jour des ensembles de clauses est faite pour éliminer les clauses subsumées et tautologies éventuelles.
- La preuve est terminée dès que l'on trouve la clause vide.

Par la suite des améliorations ont été faites.
- La première chose est la séparation des clauses. A chaque fois que l'on prend une clause candidate, on essaie de séparer cette clause en plusieurs parties, dans chacune desquelles on ajoute alors des littéraux de séparation. Une séparation n'est possible que si aucune variable libre n'est commune aux deux parties.
- Une seconde amélioration porte sur la décomposition, que l'on rend encore plus lente. C'est à dire que l'on a un ensemble de clauses décomposables, trié par poids croissant, et à chaque

étape, on prend soit une clause candidate, soit une clause décomposable selon que l'une est plus légère que l'autre.
Dans tous les cas on ne décompose que sur une étape, c'est à dire que l'on ne continue pas sur les formules obtenues après une étape de décomposition. Les décompositions sont alors ajoutées aux décomposées de la clause d'origine. Cela permet de gagner beaucoup de temps dans des preuves où il est inutile de décomposer beaucoup.

Remarque III.3.2 Le dernier point ci-dessus permet de rendre les démonstrations moins sensibles à l'ajout d'hypothèses. En effet, une des difficultés que nous avons déjà signalées est que les hypothèses sont ajoutées et ne sont jamais supprimées. On peut alors se demander si une grande quantité d'hypothèses n'engendre pas une perte d'efficacité du démonstrateur.
Le fait que l'on ne décompose que petit à petit, et que les poids permettent de faire en sorte que les hypothèses non utiles soient peu manipulées nous incite à penser que de courtes preuves ne nécessitant pas d'aller en profondeur ne sont pas perturbées par des hypothèses supplémentaires.
Il est cependant important que le poids soit effectivement bien donné, car si toutes les hypothèses ont le même poids, elles sont traitées de la même manière, et donc les hypothèses inutiles brouillent la recherche de la preuve.

III.4 Le poids des clauses

A chaque décomposition, utilisation de la résolution ou de la contraction, une clause est créée. Il faut donc donner un poids à cette clause. Celui-ci est calculé par une fonction qui dépend de plusieurs paramètres, dont les poids des clauses dont elle provient, l'unificateur, etc...
Certaines idées nous ont permis de donner une forme générale aux fonctions calculant le poids. Bien sûr il ne s'agit pas de certitudes, mais de pistes de travail. Il reste toujours à améliorer ces fonctions qui sont encore très imparfaites.

- Si les clauses parents ont un poids élevé, alors la clause obtenue doit aussi avoir un poids élevé.
- Si un unificateur est complexe (c'est à dire si la substitution a un grand domaine par exemple), alors on peut penser que le raisonnement est trop complexe. Alors le poids est croissant en la complexité de l'unificateur.
- Si la taille du littéral unifié est grande, on peut penser au contraire que le raisonnement est correct, car il est assez rare de pouvoir unifier deux formules de grande taille. Ainsi le poids est décroissant en la taille du littéral.
- Si la clause obtenue utilise trop d'hypothèses ayant un indice élevé, alors son poids doit être grand, puisqu'il y a peu de chance que ce soit un bon raisonnement.
- Si les clauses parents ont peu de littéraux, on peut considérer que c'est une bonne chose, puisque le but est d'atteindre la clause vide, et que l'obtenir avec des clauses avec beaucoup de littéraux demande *a priori* plus de raisonnement qu'avec des clauses unitaires.

Le calcul des indices est quant à lui très peu développé. Nous pouvons seulement indiquer que l'on conserve un historique de l'utilisation aussi bien des clauses que des règles. En effet, à la fois les règles et les clauses doivent bénéficier d'indices, car certaines règles ne doivent pas être utilisées sans arrêt (par exemple des règles de commutativité), et nous l'avons déjà signalé pour des hypothèses, qui sont transformées en clause.

Annexe IV

Des exemples de preuve

Nous donnons ici des exemples de preuves. Dans les deux premières nous traduirons des preuves en langue naturelle vers le langage restreint. Aucun outil n'ayant été implémenté à ce jour, les traductions que nous donnons ont été faites à la main. Ce sont donc des traductions idéales.
Ces deux preuves sont celles que nous avons déjà vues au chapitre 1. Leur traduction dans le langage restreint prend en compte ce que nous avons dit dans ce chapitre.
La troisième preuve est une preuve d'une formule très simple du premier ordre faite dans le langage restreint. L'intérêt de cette preuve est de voir le comportement du démonstrateur, dont nous afficherons les preuves.
Nous verrons ensuite d'autres preuves, utilisant le langage restreint pour la logique du premier ordre.

IV.1 Preuve 1

Redonnons la propriété et sa preuve :

Proposition 1.2.3 *Soit X_1 et X_2 des espaces topologiques. Soit f une fonction de X_1 vers X_2. Si f est continue alors, pour tout x dans X_1, f est continue en x.*
Preuve : Supposons f continue et soit x dans X_1. Posons $y = f(x)$. Soit V un voisinage de y, prouvons que son image inverse est un voisinage de x. Par définition du voisinage, soit O un ouvert inclus dans V et contenant y. Comme f est continue, $f^{-1}(O)$ est ouvert et x appartient à $f^{-1}(O)$. Comme $f^{-1}(O) \subset f^{-1}(V)$, la preuve est terminée. □

Voici une traduction. Chaque commande ou phrase du langage restreint est terminée par un point. Nous donnons après chaque phrase les buts qui sont retournés à l'utilisateur. Notons que /\ signifie $\forall$, \/ signifie $\exists$ et : signifie $\in$. Bien qu'il soit possible d'afficher ces éléments comme leurs symboles habituels, nous souhaitons montrer la syntaxe brute de la machine afin d'accentuer la différence avec la preuve naturelle.
Certains éléments dans la preuve ont un suffixe avec un numéro. C'est le cas par exemple de "voisinage_de.2" et "ouvert.1". Cela provient du fait que nous avons importé un module d'espace topologique à deux reprises pour en avoir deux, qui comportent donc les mêmes propriétés.
Pour les différencier, il faut alors utiliser ces notations avec des suffixes. C'est à dire que lors de l'importation du module d'espace topologique, nous donnons à tout élément (constante, proposition, etc...) un nouveau nom qui est le nom de l'élément dans le module suffixé avec un nombre.
Ainsi par exemple "inverse f V X.1" représente l'image réciproque par f de V dans l'espace topologique X.1, et "O ouvert.2" signifie que O est un ouvert de X.2.

```
prop f continue -> /\x:X.1 f continue_en x.
```

```
1 new goal :

|- f continue -> /\x:X.1  f continue_en x

assume f continue let x assume X.1 x show f continue_en x.

1 new goal

H := f continue
H0 := X.1 x
   |- f continue_en x

local y=f x.

0 new goal

H := f continue
H0 := X.1 x
   |- f continue_en x

let V assume V voisinage_de.2 y show (inverse f V X.1) voisinage_de.1 x.

1 new goal

H := f continue
H0 := X.1 x
H1 := V voisinage_de.2 f x
   |- inverse f V X.1 voisinage_de.1 x

by voisinage_de.2 let O assume O subset V and O ouvert.2 and O y.

1 new goal

H := f continue
H0 := X.1 x
H1 := V voisinage_de.2 f x
H2 := O subset V
H3 := O ouvert.2
H4 := O y
   |- inverse f V X.1 voisinage_de.1 x

by f continue deduce (inverse f O X.1) ouvert.1 & (inverse f O X.1) x.

1 new goal

H := f continue
H0 := X.1 x
H1 := V voisinage_de.2 f x
H2 := O subset V
H3 := O ouvert.2
H4 := O y
H5 := inverse f O X.1 ouvert.1 & inverse f O X.1 x
   |- inverse f V X.1 voisinage_de.1 x
```

```
deduce (inverse f O X.1) subset (inverse f V X.1) trivial.

0 new goal
```

Remarque IV.1.1 La traduction utilise un `local`. Nous n'avons pas défini ce mot clé dans le langage restreint, mais il est tout à fait possible de l'y inclure. Il faut cependant bien noter que son interprétation ne modifie en rien les buts courants, et qu'il ne s'agit que d'une commande administrative, qui ne fait pas avancer la preuve.

IV.2 Preuve 2

Rappelons également la propriété et sa preuve :

Proposition 1.2.4 *Pour toutes suites $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ et pour tout réel l, si $(v_n)_{n\in\mathbb{N}}$ est extraite de $(u_n)_{n\in\mathbb{N}}$ et si $(u_n)_{n\in N}$ converge vers l, alors $(v_n)_{n\in\mathbb{N}}$ converge vers l.*
Preuve : Soit v_n une suite extraite de u_n avec u_n qui converge vers l, montrons que v_n converge vers l. C'est à dire : soit $e > 0$, on cherche à prouver qu'il existe n_1 tel que pour tout $n > n_1$ $d(v_n, l) < e$. Comme v_n est une suite extraite alors il existe une fonction f croissante telle que $v_n = u_{f(n)}$ pour tout n.
Comme $e > 0$ et u_n converge vers l, alors il existe n_0 tel que pour tout $n > n0$ $d(u_n, l) < e$. Soit $n > n_0$, montrons qu'alors $d(v_n, l) < e$. Mais maintenant, si $f(n) > n_0$, comme $v_n = u_{f(n)}$, alors on aura démontré que v_n converge vers l. Comme f est croissante et que $n > n_0$, on a $f(n) > f(n_0) \geq n_0$, d'où $f(n) > n_0$. □

Voici une traduction. Notons qu'il y a deux symboles $>$ dans la preuve, `>>` et `>`. En effet l'utilisateur a défini un symbole de relation des réels différent de celui des entiers. Nous ne donnons pas ici les buts courants après chaque étape, ceci afin de permettre une meilleure lecture du texte original et de sa traduction.

```
let v_n,u_n,l  assume v_n extraite_de u_n
and u_n converge_vers l show v_n converge_vers l.
let e>>zero show \/n_1 /\n > n_1 d (v_n n) l << e.
by  v_n extraite_de u_n let f assume f croissante
and /\n (v_n n) = u_n (f n).
by e>>zero by $u_n converge_vers l$ let n_0
assume /\n>n_0 d (u_n n) l <<e.
let n assume n>n0 show d (u_n n) l << e.
prove f n > n_0 in by /\n (v_n = u_n (f n))
deduce v_n converge_vers l trivial.
by f croissante by n>n_0 deduce f n > f n_0 and f n_0 >= n_0.
deduce f n > n_0 trivial.
```

IV.3 Preuve 3

Nous donnons ici une preuve écrite dans la logique du premier ordre avec le langage restreint défini au chapitre 2. Il s'agit de la logique dont nous avons parlé au chapitre 4.
Pour simplifier l'analyse syntaxique, les formules sont écrites entre `$`. La formule que nous prouvons est évidente, mais le but est simplement de montrer le comportement du démonstrateur, en affichant la preuve de celui-ci.

Pour chaque règle qui nécessite d'être validée, on fait appel au démonstrateur, et si le but qui lui a été donné est prouvé, le démonstrateur écrit la mention "La formule est prouvée", et affiche la preuve.

Il est possible de savoir quelle formule a été prouvée en regardant la dernière formule affichée : il s'agit de la négation (~) de la formule prouvée.

L'affichage de la preuve se fait par une arborescence, la règle de résolution étant binaire, et toutes les autres règles étant linéaire. Ce qui est affiché est donc l'arbre de preuve associé à la preuve par résolution.

Nous incitons le lecteur à lire le chapitre 2 pour comprendre les buts qui sont donnés au démonstrateur.
Néanmoins pour la seconde phrase il y a eu trois buts à prouver. En effet, le premier est la justification de `let a assume $p(a)$ show $q(a)$`. Le second but est la preuve que l'on peut déduire $q(a)$, et le dernier termine la preuve par `trivial`.

Les notations sont les suivantes (voir l'annexe III pour plus d'informations) :
`(35)` est le numéro d'une clause. Il ne faut cependant pas s'inquiéter de la grandeur de ces nombres car de nouveaux numéros sont donnés à chaque nouvelle clause créée même après qu'une preuve soit terminée.
`#` donne le poids de la clause
`%` donne l'indice de la clause
`$dec` indique que la clause provient d'une décomposition et donne la règle et la formule
`$res` indique que la règle de résolution a été appliquée, en donnant le littéral.

```
>>> goal $(/\x.(p(x)->q(x)))->/\a.(p(a)->q(a))$
1 new goal

|- /\ x. (p(x) -> q(x)) -> /\ a. (p(a) -> q(a))

>>> assume $/\x.(p(x)->q(x))$ show $/\a.(p(a)->q(a))$
La formule est vraie
Voici la preuve :
\(23) :  #0. %4. $res /\ x. (p(x) -> q(x)) -> /\ a. (p(a) -> q(a))
 \(21)/\ x. (p(x) -> q(x)) -> /\ a. (p(a) -> q(a)) :  #1. %1.
 \(22)~(/\ x. (p(x) -> q(x)) -> /\ a. (p(a) -> q(a))) :  #1. %2.
1 new goal

 :/\ x. (p(x) -> q(x))
|- /\ a. (p(a) -> q(a))

>>> let a assume $p(a)$ show $q(a)$ deduce $q(a)$ trivial shown
La formule est vraie
Voici la preuve :
\(27) :  #0. %4. $res /\ a. (p(a) -> q(a))
 \(24)/\ a. (p(a) -> q(a)) :  #1. %1.
 \(26)~/\ a. (p(a) -> q(a)) :  #1. %2.
La formule est vraie
Voici la preuve :
\(35) :  #0. %7. $res q(a)
 \(33)q(a) :  #5.4 %3. $res p(a)
 |\(28)p(a) :  #1. %1.
 |\(32)q(x), ~p(x) :  #3.4 %1. $dec +imp p(x) -> q(x)
 | \(31)p(x) -> q(x) :  #2.3 %1. $dec +fa /\ x. (p(x) -> q(x))
 |  \(29)/\ x. (p(x) -> q(x)) :  #1. %1.
 \(30)~q(a) :  #1. %2.
La formule est vraie
Voici la preuve :
```

```
\(40) :  #0. %4. $res q(a)
 \(36)q(a) :  #1. %1.
 \(39)~q(a) :  #1. %2.
Rien a prouver
```

Remarque IV.3.1 Le lecteur peut avoir remarqué une différence dans l'affichage du but courant entre la première preuve et la dernière. Les hypothèses ne sont pas nommées dans cette dernière alors qu'elles le sont dans la première. Ceci est pour mettre en avant le fait que la machine peut avoir besoin d'un nom, alors que l'utilisateur n'en donne pas.
Dans la logique du premier ordre avec langage restreint implémenté nous avons décidé de ne pas afficher de nom à des hypothèses non nommées par l'utilisateur. Cela explique ainsi que dans cette dernière preuve il n'y a aucun nom dans les buts.

IV.4 Preuve 4

Proposition $\forall f, g((f \circ g \ \textit{injective} \vee f \circ g \ \textit{surjective}) \rightarrow (g \ \textit{injective} \vee f \ \textit{surjective}))$.
Preuve : Soit f et g.
- Supposons que $f \circ g$ est injective. Soit x et y tels que $g(x) = g(y)$. Alors $f \circ g(x) = f \circ g(y)$. Par injectivité de $f \circ g$, $x = y$ et donc g est injective.
- Supposons maintenant que $f \circ g$ est surjective. Soit z, alors il existe x tel que $f \circ g(x)$ est égal à z. Par conséquent $f(g(x)) = z$. Donc f est surjective. □

Voici comment faire cela dans le système de la logique du premier ordre avec le langage restreint. Comme la logique ne gère pas l'égalité, il est nécessaire d'ajouter des axiomes, qui sont transformés en un ensemble de clauses par le démonstrateur.

```
axiom $/\f. (inj(f) <-> /\x,y. (eq(app(f,x),app(f,y))-> eq(x,y)))$
axiom $/\f. surj(f) <-> /\y. \/x. eq(app(f,x),y)$
axiom $/\f,g. /\x. eq(app(rond(f,g),x),app(f,app(g,x)))$

axiom $/\x,y,z. (eq(x,y) -> eq (y,z) -> eq(x,z))$
axiom $/\x,y. (eq(x,y) -> eq(y,x))$
axiom $/\x. eq(x,x)$

axiom $/\f,x,y. (eq(x,y) -> eq(app(f,x),app(f,y)))$

goal $/\f,g. ((inj(rond(f,g)) or surj(rond(f,g))) -> (surj(f) or inj(g)))$
let f,g begin assume $inj(rond(f,g))$ show $inj(g)$
then assume $surj(rond(f,g))$ show $surj(f)$ end
let x,y assume $eq(app(g,x),app(g,y))$ show $eq(x,y)$
deduce $eq(app(rond(f,g),x),app(rond(f,g),y))$
by $inj(rond(f,g))$ deduce $eq(x,y)$ trivial
let z show $\/x. eq(app(f,x),z)$
let x assume $eq(app(rond(f,g),x),z)$
deduce $eq(app(f,app(g,x)),z)$ trivial
```

Voici la preuve en une seule commande, qui est validée de la même manière après interprétation :

```
goal $/\f,g. ((inj(rond(f,g)) or surj(rond(f,g))) -> (surj(f) or inj(g)))$
let f,g
begin assume $inj(rond(f,g))$
  show $inj(g)$
    let x,y assume $eq(app(g,x),app(g,y))$
    show $eq(x,y)$
      deduce $eq(app(rond(f,g),x),app(rond(f,g),y))$
```

```
      by $inj(rond(f,g))$ deduce $eq(x,y)$ trivial
    shown
  shown
then
  assume $surj(rond(f,g))$
  show $surj(f)$
    let z
      show $\/x. eq(app(f,x),z)$
        let x assume $eq(app(rond(f,g),x),z)$
        proof
          deduce $eq(app(f,app(g,x)),z)$ trivial
        endproof
      shown
  shown
end
```

On peut penser que cette preuve en une seule commande (qui normalement n'est qu'une seule ligne de texte), bien que plus complexe, retrace plus une analyse globale de la preuve, qui est faite par cas. Les cas étant donnés au fur et à mesure, ce n'est qu'en ayant analysé toute la preuve que l'on sait sur quels cas se base l'utilisateur.
On remarque également qu'ici seuls quelques mots clés supplémentaires ont été ajoutés à une mise bout à bout des commandes données dans la preuve en plusieurs étapes. La preuve reste donc assez bien lisible, surtout si on la structure. Notons cependant qu'il n'est pas un objectif que de telles preuves soient lisibles, puisque l'utilisateur final n'a normalement pas accès à ce langage, qui reste un outil dans l'analyse de la preuve.

IV.5 Preuve 5

Deux exercices de topologie sur les points adhérents, isolés et d'accumulation. Tout d'abord nous devons donner les définitions de ces trois notions.

```
axiom $/\A,x. (adh(A,x)<->
          (/\e. (less(zero(),e) -> \/y. (app(A,y) & less(d(x,y),e)))))$
axiom $/\A,x. (isol(A,x)<->
          (app(A,x) & \/e. (less(zero(),e)
                              & /\y. ((less(d(x,y),e) & app(A,y))
                                        -> eq (y,x)))))$
axiom $/\A,x. (acc(A,x)<->
          /\e. (less(zero(),e)
                 -> \/y. (app(A,y) & (less(d(x,y),e) & ~eq(y,x)))))$
```

Voici un premier exercice, qui prouve qu'un point ne peut pas être à la fois un point isolé et un point d'accumulation.

```
goal $/\A,x. ~(isol(A,x) & acc(A,x))$
let A,x assume $isol(A,x)$ [Is] and $acc(A,x)$ [Ac] show $false$
by [Is] deduce $app(A,x)$
let e assume $less(zero(),e)$
        and $/\y. (less(d(x,y),e) & app(A,y)-> eq(y,x))$ [E]
by [Ac] let y assume $app(A,y)$ and $less(d(x,y),e)$ and $~(eq(y,x))$
by [E] deduce $eq(y,x)$ trivial
```

Avant de poursuivre, nous devons encore ajouter des propriétés permettant de traiter l'égalité.

```
axiom $/\x. (eq(d(x,x),zero()))$
axiom $/\x,y,z. ((eq(x,y) & less(y,z))-> less(x,z))$
axiom $/\A. /\x,y. (eq(y,x) -> app(A,y) -> app(A,x))$
```

Voici enfin le second exercice qui démontre qu'un point adhérent est soit un point isolé, soit un point d'accumulation.

```
goal $/\x,A. (adh(A,x) <-> (isol(A,x) or acc(A,x)))$
let x,A begin assume $adh(A,x)$ show $isol(A,x) or acc(A,x)$
           then assume $isol(A,x)$ show $adh(A,x)$
           then assume $acc(A,x)$ show $adh(A,x)$ end
assume $acc(A,x)$ proof trivial endproof
  then assume $~ acc(A,x)$ show $isol(A,x)$
let e assume $less(zero(),e)$
  and $/\y. (app(A,y) -> (~ less(d(x,y),e) or eq(y,x)))$ [Eq]
by $adh(A,x)$ let y assume $app(A,y)$ and $less(d(x,y),e)$
by [Eq] with $app(A,y)$ deduce $eq(y,x)$
trivial
let e assume $less(zero(),e)$ show $less(d(x,x),e)$ trivial shown
by $acc(A,x)$ trivial
```

On peut remarquer ici une preuve se faisant avec trois cas : la première commande contient une succession de deux `then`.
On voit également d'utilisation du `by`, avec le nom des hypothèses ou avec les formules.

Annexe V

Preuves du chapitre 5

Nous donnons ici les parties de preuve des propositions du chapitre 5 que nous avons laissées inachevées.

Le lecteur est incité à lire les énoncés et le début des preuves des propositions pour comprendre la situation. Nous donnons ici seulement les cas qu'il reste à prouver.

V.1 Proposition 5.4.4

– $R = \vee_{n_g} : S$ contient une clause $\Gamma, (A \vee B)^{\perp}$ et $C = \Gamma, A^{\perp}$. Nous avons donc par hypothèse $\vdash_{LK} \Sigma, \Pi \wedge (A \vee B), \Pi \wedge A$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi \vdash \Pi}\,ax}{\Pi \wedge A \vdash \Pi}\,\wedge_{g_d} \qquad \vdash \Sigma, \Pi \wedge (A \vee B), \Pi \wedge A}{\vdash \Sigma, \Pi \wedge (A \vee B), \Pi}\,cut \qquad \dfrac{\dfrac{\dfrac{}{A \vdash A}\,ax}{\Pi \wedge A \vdash A}\,\wedge_{g_g} \qquad \vdash \Sigma, \Pi \wedge (A \vee B), \Pi \wedge A}{\vdash \Sigma, \Pi \wedge (A \vee B), A}\,cut}{\vdash \Sigma, \Pi \wedge (A \vee B), A \vee B}\,\vee_{d_g}}{\vdash \Sigma, \Sigma, \Pi \wedge (A \vee B), \Pi \wedge (A \vee B), \Pi \wedge (A \vee B)}\,\wedge_d}{\vdash \Sigma, \Pi \wedge (A \vee B)}\,c_d^*$$

– $R = \vee_{n_d} : S$ contient une clause $\Gamma, (A \vee B)^{\perp}$ et $C = \Gamma, B^{\perp}$. Nous avons donc par hypothèse $\vdash_{LK} \Sigma, \Pi \wedge (A \vee B), \Pi \wedge B$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi \vdash \Pi}\,ax}{\Pi \wedge B \vdash \Pi}\,\wedge_{g_d} \qquad \vdash \Sigma, \Pi \wedge (A \vee B), \Pi \wedge B}{\vdash \Sigma, \Pi \wedge (A \vee B), \Pi}\,cut \qquad \dfrac{\dfrac{\dfrac{}{B \vdash B}\,ax}{\Pi \wedge B \vdash B}\,\wedge_{g_g} \qquad \vdash \Sigma, \Pi \wedge (A \vee B), \Pi \wedge B}{\vdash \Sigma, \Pi \wedge (A \vee B), B}\,cut}{\vdash \Sigma, \Pi \wedge (A \vee B), A \vee B}\,\vee_{d_d}}{\vdash \Sigma, \Sigma, \Pi \wedge (A \vee B), \Pi \wedge (A \vee B), \Pi \wedge (A \vee B)}\,\wedge_d}{\vdash \Sigma, \Pi \wedge (A \vee B)}\,c_d^*$$

– $R = \rightarrow_p : S$ contient une clause $\Gamma, A \rightarrow B$ et $C = \Gamma, A^{\perp}, B$. Nous avons donc par hypothèse $\vdash_{LK} \Sigma, \Pi \wedge \neg(A \rightarrow B), \Pi \wedge A \wedge \neg B$. Définissons T comme

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,ax}{\Pi \wedge A \vdash A}\,\wedge_{g_g}}{\Pi \wedge A \wedge \neg B \vdash A}\,\wedge_{g_d} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{B \vdash B}\,ax}{\neg B, B \vdash}\,\neg_g}{A \wedge \neg B, B \vdash}\,\wedge_{g_g}}{\Pi \wedge A \wedge \neg B, B \vdash}\,\wedge_{g_g}}{\Pi \wedge A \wedge \neg B, \Pi \wedge A \wedge \neg B, A \to B \vdash}\,\to_g}{\Pi \wedge A \wedge \neg B, A \to B \vdash}\,c_g}{\Pi \wedge \neg A \wedge \neg B \vdash \neg(A \to B)}\,\neg_d \qquad \vdash \Sigma, \Pi \wedge \neg(A \to B), \Pi \wedge \neg A \wedge \neg B}{\vdash \Sigma, \Pi \wedge \neg(A \to B), \neg(A \to B)}\,cut$$

Alors

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi \vdash \Pi}\,ax}{\Pi \wedge A \vdash \Pi}\,\wedge_{g_d}}{\Pi \wedge A \wedge \neg B \vdash \Pi}\,\wedge_{g_d} \qquad \vdash \Sigma, \Pi \wedge \neg(A \to B), \Pi \wedge A \wedge \neg B}{\vdash \Sigma, \Pi \wedge \neg(A \to B), \Pi}\,cut \qquad T}{\vdash \Sigma, \Sigma, \Pi \wedge \neg(A \to B), \Pi \wedge \neg(A \to B), \Pi \wedge \neg(A \to B)}\,\wedge_d}{\vdash \Sigma, \Pi \wedge \neg(A \to B)}\,c^*_d$$

– $R = \to_{n_g} : S$ contient une clause $\Gamma, (A \to B)^\perp$ et $C = \Gamma, A$. Nous avons donc par hypothèse $\vdash_{LK} \Sigma, \Pi \wedge (A \to B), \Pi \wedge \neg A$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi \vdash \Pi}\,ax}{\Pi \wedge \neg A \vdash \Pi}\,\wedge_{g_d} \qquad \vdash \Sigma, \Pi \wedge (A \to B), \Pi \wedge \neg A}{\vdash \Sigma, \Pi \wedge (A \to B), \Pi}\,cut \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,ax}{\neg A, A \vdash}\,\neg_g}{\Pi \wedge \neg A, A \vdash}\,\wedge_{g_g} \qquad \vdash \Sigma, \Pi \wedge (A \to B), \Pi \wedge \neg A}{A \vdash \Sigma, \Pi \wedge (A \to B)}\,cut}{\vdash \Sigma, \Pi \wedge (A \to B), A \to B}\,\to_{d_g}}{\vdash \Sigma, \Sigma, \Pi \wedge (A \to B), \Pi \wedge (A \to B), \Pi \wedge (A \to B)}\,\wedge_d}{\vdash \Sigma, \Pi \wedge (A \to B)}\,c^*_d$$

– $R = \to_{n_d} : S$ contient une clause $\Gamma, (A \to B)^\perp$ et $C = \Gamma, B^\perp$. Nous avons donc par hypothèse $\vdash_{LK} \Sigma, \Pi \wedge (A \to B), \Pi \wedge B$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi \vdash \Pi}\,ax}{\Pi \wedge B \vdash \Pi}\,\wedge_{g_d} \qquad \vdash \Sigma, \Pi \wedge (A \to B), \Pi \wedge B}{\vdash \Sigma, \Pi \wedge (A \to B), \Pi}\,cut \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{B \vdash B}\,ax}{\Pi \wedge B \vdash B}\,\wedge_{g_g} \qquad \vdash \Sigma, \Pi \wedge (A \to B), \Pi \wedge B}{\vdash \Sigma, \Pi \wedge (A \to B), B}\,cut}{\vdash \Sigma, \Pi \wedge (A \to B), A \to B}\,\to_{d_d}}{\vdash \Sigma, \Sigma, \Pi \wedge (A \to B), \Pi \wedge (A \to B), \Pi \wedge (A \to B)}\,\wedge_d}{\vdash \Sigma, \Pi \wedge (A \to B)}\,c^*_d$$

– $R = \wedge_n : S$ contient une clause $\Gamma, (A \wedge B)^\perp$ et $C = \Gamma, A^\perp, B^\perp$. Nous avons donc par hypothèse $\vdash_{LK} \Sigma, \Pi \wedge (A \wedge B), \Pi \wedge A \wedge B$. Définissons T comme

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,ax}{\Pi \wedge A \vdash A}\,\wedge_{g_g}}{\Pi \wedge A \wedge B \vdash A}\,\wedge_{g_d} \quad \dfrac{\dfrac{\dfrac{}{B \vdash B}\,ax}{A \wedge B \vdash B}\,\wedge_{g_g}}{\Pi \wedge A \wedge B \vdash B}\,\wedge_{g_g}}{\Pi \wedge A \wedge B, \Pi \wedge A \wedge B \vdash A \wedge B}\,\wedge_d}{\Pi \wedge A \wedge B \vdash A \wedge B}\,c_g \qquad \vdash \Sigma, \Pi \wedge (A \wedge B), \Pi \wedge A \wedge B}{\vdash \Sigma, \Pi \wedge (A \wedge B), A \wedge B}\,cut$$

Alors

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi \vdash \Pi}\,ax}{\Pi \wedge A \vdash \Pi}\,\wedge_{g_d}}{\Pi \wedge A \wedge B \vdash \Pi}\,\wedge_{g_d} \qquad \vdash \Sigma, \Pi \wedge (A \wedge B), \Pi \wedge A \wedge B}{\dfrac{\dfrac{\vdash \Sigma, \Pi \wedge (A \wedge B), \Pi \qquad T}{\vdash \Sigma, \Sigma, \Pi \wedge (A \wedge B), \Pi \wedge (A \wedge B), \Pi \wedge (A \wedge B)}\,\wedge_d}{\vdash \Sigma, \Pi \wedge (A \wedge B)}\,c_d^*}\,cut$$

– $R = \wedge_{p_g}$: S contient une clause $\Gamma, A \wedge B$ et $C = \Gamma, A$. Nous avons donc par hypothèse $\vdash_{LK} \Sigma, \Pi \wedge \neg(A \wedge B), \Pi \wedge \neg A$. Définissons T comme

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,ax}{\neg A, A \vdash}\,\neg_g}{\Pi \wedge \neg A, A \vdash}\,\wedge_{g_g}}{\Pi \wedge \neg A, A \wedge B \vdash}\,\wedge_{g_d}}{\Pi \wedge \neg A \vdash \neg(A \wedge B)}\,\neg_d \qquad \vdash \Sigma, \Pi \wedge \neg(A \wedge B), \Pi \wedge \neg A}{\vdash \Sigma, \Pi \wedge \neg(A \wedge B), \neg(A \wedge B)}\,cut$$

Alors

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi \vdash \Pi}\,ax}{\Pi \wedge \neg A \vdash \Pi}\,\wedge_{g_d}}{\Pi \wedge \neg A \vdash \Pi}\,cut \qquad \vdash \Sigma, \Pi \wedge \neg(A \wedge B), \Pi \wedge \neg A}{\dfrac{\vdash \Sigma, \Pi \wedge \neg(A \wedge B), \Pi \qquad T}{\vdash \Sigma, \Sigma, \Pi \wedge \neg(A \wedge B), \Pi \wedge \neg(A \wedge B), \Pi \wedge \neg(A \wedge B)}\,\wedge_d}{\vdash \Sigma, \Pi \wedge \neg(A \wedge B)}\,c_d^*$$

– $R = \wedge_{p_d}$: S contient une clause $\Gamma, A \wedge B$ et $C = \Gamma, B$. Nous avons donc par hypothèse $\vdash_{LK} \Sigma, \Pi \wedge \neg(A \wedge B), \Pi \wedge \neg B$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{B \vdash B}\,ax}{\neg B, B \vdash}\,\neg_g}{\Pi \wedge \neg B, B \vdash}\,\wedge_{g_g}}{\Pi \wedge \neg B, A \wedge B \vdash}\,\wedge_{g_g}}{\Pi \wedge \neg B \vdash \neg(A \wedge B)}\,\neg_d \qquad \vdash \Sigma, \Pi \wedge \neg(A \wedge B), \Pi \wedge \neg B}{\vdash \Sigma, \Pi \wedge \neg(A \wedge B), \neg(A \wedge B)}\,cut$$

Alors

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi \vdash \Pi}\,ax}{\Pi \wedge \neg B \vdash \Pi}\,\wedge_{g_d} \qquad \vdash \Sigma, \Pi \wedge \neg(A \wedge B), \Pi \wedge \neg B}{\vdash \Sigma, \Pi \wedge \neg(A \wedge B), \Pi}\,cut \qquad T}{\vdash \Sigma, \Sigma, \Pi \wedge \neg(A \wedge B), \Pi \wedge \neg(A \wedge B), \Pi \wedge \neg(A \wedge B)}\,\wedge_d}{\vdash \Sigma, \Pi \wedge \neg(A \wedge B)}\,c_d^*$$

– $R = \neg_n : S$ contient une clause $\Gamma, (\neg A)^\perp$ et $C = \Gamma, A$. Nous avons donc par hypothèse $\vdash_{LK} \Sigma, \Pi \wedge (\neg A), \Pi \wedge \neg A$, ce qui par contraction nous donne le résultat souhaité :

$$\dfrac{\vdash \Sigma, \Pi \wedge \neg A, \Pi \wedge \neg A}{\vdash \Sigma, \Pi \wedge \neg A}\,c_d$$

– $R = \neg_p : S$ contient une clause $\Gamma, \neg A$ et $C = \Gamma, A^\perp$. Nous avons donc par hypothèse $\vdash_{LK} \Sigma, \Pi \wedge \neg(\neg A), \Pi \wedge A$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi \vdash \Pi}\,ax}{\Pi \wedge A \vdash \Pi}\,\wedge_{g_d} \qquad \vdash \Sigma, \Pi \wedge \neg(\neg A), \Pi \wedge A}{\vdash \Sigma, \Pi \wedge \neg(\neg A), \Pi}\,cut \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,ax}{A, \neg A \vdash}\,\neg_g}{A \vdash \neg(\neg A)}\,\neg_d}{\Pi \wedge A \vdash \neg(\neg A)}\,\wedge_{g_g} \qquad \vdash \Sigma, \Pi \wedge \neg(\neg A), \Pi \wedge A}{\vdash \Sigma, \Pi \wedge \neg(\neg A), \neg(\neg A)}\,cut}{\vdash \Sigma, \Pi \wedge \neg(\neg A), \Pi \wedge \neg(\neg A)}\,\wedge_d}{\vdash \Sigma, \Pi \wedge \neg(\neg A)}\,c_d$$

– $R = \exists_p : S$ contient une clause $\Gamma, \exists x.A(x)$ et $C = \Gamma, A(y)$ avec y une variable fraîche. Nous avons donc par hypothèse $\vdash_{LK} \Sigma, \Pi \wedge \neg\exists x.A(x), \Pi \wedge \neg A(y)$. Définissons T comme

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A(y) \vdash A(y)}\,ax}{\neg A(y), A(y) \vdash}\,\neg_g}{\Pi \wedge \neg A(y), A(t) \vdash}\,\wedge_{g_g} \qquad \vdash \Sigma, \Pi \wedge \neg\exists x.A(x), \Pi \wedge \neg A(y)}{A(y) \vdash \Sigma, \Pi \wedge \neg\exists x.A(x)}\,cut}{\exists x.A(x) \vdash \Sigma, \Pi \wedge \neg\exists x.A(x)}\,\exists_g(*)}{\vdash \Sigma, \Pi \wedge \neg\exists x.A(x), \neg\exists x.A(x)}\,\neg_d$$

Alors

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi \vdash \Pi}\,ax}{\Pi \wedge A(y) \vdash \Pi}\,\wedge_{g_d} \qquad \vdash \Sigma, \Pi \wedge \neg\exists x.A(x), \Pi \wedge A(y)}{\vdash \Sigma, \Pi \wedge \neg\exists x.A(x), \Pi}\,cut \qquad T}{\vdash \Sigma, \Sigma, \Pi \wedge \neg\exists x.A(x), \Pi \wedge \neg\exists x.A(x), \Pi \wedge \neg\exists x.A(x)}\,\wedge_d}{\vdash \Sigma, \Pi \wedge \neg\exists x.A(x)}\,c_d^*$$

(*) la condition est bien remplie.

– $R = \exists_n : S$ contient une clause $\Gamma, (\exists x.A(x))^\perp$ et $C = \Gamma, A(t)^\perp$. Nous avons donc par hypothèse $\vdash_{LK} \Sigma, \Pi \wedge \exists x.A(x), \Pi \wedge A(t)$. Définissons T comme

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A(t) \vdash A(t)}\,ax}{\Pi \wedge A(t) \vdash A(t)}\,\wedge_{g_g} \qquad \vdash \Sigma, \Pi \wedge \exists x.A(x), \Pi \wedge A(t)}{\vdash \Sigma, \Pi \wedge \exists x.A(x), A(t)}\,cut}{\vdash \Sigma, \Pi \wedge \exists x.A(x), \exists x.A(x)}\,\exists_d$$

Alors

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi \vdash \Pi}\,ax}{\Pi \wedge A(t) \vdash \Pi}\,\wedge_{g_d} \qquad \vdash \Sigma, \Pi \wedge \exists x.A(x), \Pi \wedge A(t)}{\vdash \Sigma, \Pi \wedge \exists x.A(x), \Pi}\,cut \qquad T}{\vdash \Sigma, \Sigma, \Pi \wedge \exists x.A(x), \Pi \wedge \exists x.A(x), \Pi \wedge \exists x.A(x)}\,\wedge_d}{\vdash \Sigma, \Pi \wedge \exists x.A(x)}\,c_d^*$$

– $R = contr_p : S$ contient une clause Γ, A, A et $C = \Gamma, A$. Nous avons donc par hypothèse $\vdash_{LK} \Sigma, \Pi \wedge \neg A \wedge \neg A, \Pi \wedge \neg A$. Définissons T comme

$$\dfrac{\dfrac{\dfrac{}{\neg A \vdash \neg A}\,ax}{\Pi \wedge \neg A \vdash \neg A}\,\wedge_{g_g} \qquad \vdash \Sigma, \Pi \wedge \neg A \wedge \neg A, \Pi \wedge \neg A}{\vdash \Sigma, \Pi \wedge \neg A \wedge \neg A, \neg A}\,cut$$

Alors

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi \vdash \Pi}\,ax}{\Pi \wedge \neg A \vdash \Pi}\,\wedge_{g_d} \qquad \vdash \Sigma, \Pi \wedge \neg A \wedge \neg A, \Pi \wedge \neg A}{\vdash \Sigma, \Pi \wedge \neg A \wedge \neg A), \Pi}\,cut \qquad \dfrac{T \qquad T}{\vdash \Sigma, \Sigma, \Pi \wedge \neg A \wedge \neg A, \Pi \wedge \neg A \wedge \neg A, \neg A \wedge \neg A}\,\wedge_d}{\vdash \Sigma, \Sigma, \Sigma, \Pi \wedge \neg A \wedge \neg A, \Pi \wedge \neg A \wedge \neg A, \Pi \wedge \neg A \wedge \neg A, \Pi \wedge \neg A \wedge \neg A}\,\wedge_d}{\vdash \Sigma, \Pi \wedge \neg A \wedge \neg A}\,c_d^*$$

– $R = contr_n : S$ contient une clause $\Gamma, A^\perp, A^\perp$ et $C = \Gamma, A^\perp$. Nous avons donc par hypothèse $\vdash_{LK} \Sigma, \Pi \wedge A \wedge A, \Pi \wedge A$. Définissons T comme

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,ax}{\Pi \wedge A \vdash A}\,\wedge_{g_g} \qquad \vdash \Sigma, \Pi \wedge A \wedge A, \Pi \wedge A}{\vdash \Sigma, \Pi \wedge A \wedge A, A}\,cut \qquad \dfrac{\dfrac{}{A \vdash A}\,ax}{\Pi \wedge A \vdash A}\,\wedge_{g_g} \qquad \vdash \Sigma, \Pi \wedge A \wedge A, \Pi \wedge A}{\vdash \Sigma, \Pi \wedge A \wedge A, A}\,cut}{\vdash \Sigma, \Sigma, \Pi \wedge A \wedge A, \Pi \wedge A \wedge A, A \wedge A}\,\wedge_d$$

Alors

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi \vdash \Pi}\,ax}{\Pi \wedge A \vdash \Pi}\,\wedge_{g_d} \qquad \vdash \Sigma, \Pi \wedge A \wedge A, \Pi \wedge A}{\vdash \Sigma, \Pi \wedge A \wedge A), \Pi}\,cut \qquad T}{\vdash \Sigma, \Sigma, \Sigma, \Pi \wedge A \wedge A, \Pi \wedge A \wedge A, \Pi \wedge A \wedge A, \Pi \wedge A \wedge A}\,\wedge_d}{\vdash \Sigma, \Pi \wedge A \wedge A}\,c_d^*$$

– $R = res : S$ contient une clause $\Gamma^\perp, \Delta, A$ et une clause $\Gamma'^\perp, \Delta', A^\perp$ et $C = \Gamma^\perp, \Gamma^\perp, \Delta, \Delta'$. Nous avons donc par hypothèse $\vdash_{LK} \Sigma, \Pi \wedge \neg A, \Pi' \wedge A, \Pi \wedge \Pi'$. Définissons T comme

$$\dfrac{\dfrac{}{A \vdash A}\,ax}{\dfrac{}{\vdash \neg A, A}\,\neg_d} \qquad \dfrac{\dfrac{\dfrac{}{\Pi \vdash \Pi}\,ax}{\Pi \wedge \Pi' \vdash \Pi}\,\wedge_{g_d} \qquad \vdash \Sigma, \Pi \wedge \neg A, \Pi' \wedge A, \Pi \wedge \Pi'}{\vdash \Sigma, \Pi \wedge \neg A, \Pi' \wedge A, \Pi}\,cut}{\vdash \Sigma, \Pi \wedge \neg A, \Pi' \wedge A, \Pi \wedge \neg A, A}\,\wedge_d$$

Alors

$$\dfrac{\dfrac{\dfrac{}{A \vdash A}\,ax \qquad \dfrac{\dfrac{\dfrac{}{\Pi' \vdash \Pi'}\,ax}{\Pi \wedge \Pi' \vdash \Pi'}\,\wedge_{g_d} \qquad \vdash \Sigma, \Pi \wedge \neg A, \Pi' \wedge A, \Pi \wedge \Pi'}{\vdash \Sigma, \Pi \wedge \neg A, \Pi' \wedge A, \Pi'}\,cut}{A \vdash \Sigma, \Pi \wedge \neg A, \Pi' \wedge A, \Pi' \wedge A}\,\wedge_d \qquad T}{\vdash \Sigma, \Sigma, \Pi \wedge \neg A, \Pi \wedge \neg A, \Pi \wedge \neg A, \Pi' \wedge A, \Pi' \wedge A, \Pi' \wedge A}\,cut}{\vdash \Sigma, \Pi \wedge \neg A, \Pi' \wedge A}\,c_d^*$$

□

V.2 Proposition 5.4.5

- $\dfrac{}{A \vdash A}\,ax$: Alors par la règle de résolution $\dfrac{S = A; A^\perp}{S; \square}\,res$
- $\dfrac{\Gamma, A \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\,\wedge_{g_d}$: Par hypothèse d'induction, il existe une dérivation de la clause vide à partir de $S_1; A$, S_1 représentant l'ensemble des clauses pour Γ et Δ (ce sera la même chose dans les cas suivants). Or par la règle $\wedge_{p_g}$ nous obtenons l'ensemble de clauses $S_1; A; A \wedge B$ à partir de $S_1; A \wedge B$ et nous concluons grâce au lemme 5.3.4.
- $\dfrac{\Gamma, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\,\wedge_{g_g}$: Par hypothèse d'induction, il existe une dérivation de la clause vide à partir de $T = S_1; B$. Or par la règle $\wedge_{p_d}$ nous obtenons l'ensemble de clauses $S_1; B; A \wedge B$ à partir de $S_1; A \wedge B$ et nous concluons grâce au lemme 5.3.4.
- $\dfrac{\Gamma \vdash A, \Delta \quad \Gamma' \vdash B, \Delta'}{\Gamma, \Gamma' \vdash A \wedge B, \Delta, \Delta'}\,\wedge_d$: Par hypothèse d'induction, il existe une dérivation de la clause vide à partir de $S_1; A^\perp$ et à partir de $S_1'; B^\perp$. Or par la règle $\wedge_n$ nous obtenons l'ensemble de clauses $S_1; S_1'; A^\perp, B^\perp; (A \wedge B)^\perp$ à partir de $S_1; S_1'; (A \wedge B)^\perp$. Par le lemme 5.3.4, il existe une dérivation de la clause vide à partir de $S_1; S_1'; A^\perp; (A \wedge B)^\perp$. Si la dérivation n'utilise pas la clause $A^\perp$ alors c'est fini. Sinon par le lemme 5.3.10 il existe une dérivation de la clause $B^\perp$ à partir de $S_1; S_1'; A^\perp, B^\perp; (A \wedge B)^\perp$. Une seconde utilisation du lemme 5.3.4 nous donne le résultat grâce à la dérivation de la clause vide à partir de $S_1'; B^\perp$.
- $\dfrac{\Gamma, A \vdash \Delta \quad \Gamma', B \vdash \Delta'}{\Gamma, \Gamma', A \vee B \vdash \Delta, \Delta'}\,\vee_g$: Par hypothèse d'induction, il existe une dérivation de la clause vide à partir de $S_1; A$ et à partir de $S_1'; B$. Or par la règle $\vee_p$ nous obtenons l'ensemble de clauses $S_1; S_1'; A, B; A \vee B$ à partir de $S_1; S_1'; A \vee B$. Par le lemme 5.3.4, il existe une dérivation de la clause vide à partir de $S_1; S_1'; A; A \vee B$. Si la dérivation n'utilise pas la clause A alors c'est fini. Sinon par le lemme 5.3.10 il existe une dérivation de la clause B à partir de $S_1; S_1'; A, B; A \vee B$. Une seconde utilisation du lemme 5.3.4 nous donne le résultat grâce à la dérivation de la clause vide à partir de $S_1'; B$.
- $\dfrac{\Gamma \vdash A, \Delta}{\Gamma \vdash A \vee B, \Delta}\,\vee_{d_g}$: Par hypothèse d'induction, il existe une dérivation de la clause vide à partir de $S_1; A^\perp$. Or par la règle $\vee_{n_g}$ nous obtenons l'ensemble de clauses $S_1; A^\perp; (A \vee B)^\perp$ à partir de $S_1; (A \vee B)^\perp$ et nous concluons grâce au lemme 5.3.4.

– $\dfrac{\Gamma \vdash B, \Delta}{\Gamma \vdash A \vee B, \Delta} \vee_{d_d}$: Par hypothèse d'induction, il existe une dérivation de la clause vide à partir de $S_1; B^\perp$. Or par la règle $\vee_{n_d}$ nous obtenons l'ensemble de clauses $S_1; B^\perp; (A \vee B)^\perp$ à partir de $S_1; (A \vee B)^\perp$ et nous concluons grâce au lemme 5.3.4.

– $\dfrac{\Gamma, B \vdash \Delta \quad \Gamma' \vdash A, \Delta'}{\Gamma, \Gamma', A \to B \vdash \Delta, \Delta'} \to_g$: Par hypothèse d'induction, il existe une dérivation de la clause vide à partir de $S_1; A^\perp$ et à partir de $S_1'; B$. Or par la règle $\to_p$ nous obtenons l'ensemble de clauses $S_1; S_1'; A^\perp, B; A \to B$ à partir de $S_1; S_1'; A \to B$. Par le lemme 5.3.4, il existe une dérivation de la clause vide à partir de $S_1; S_1'; A^\perp; A \to B$. Si la dérivation n'utilise pas la clause $A^\perp$ alors c'est fini. Sinon par le lemme 5.3.10 il existe une dérivation de la clause B à partir de $S_1; S_1'; A^\perp, B; A \to B$. Une seconde utilisation du lemme 5.3.4 nous donne le résultat grâce à la dérivation de la clause vide à partir de $S_1'; B$.

– $\dfrac{\Gamma, A \vdash \Delta}{\Gamma \vdash A \to B, \Delta} \to_{d_g}$: Par hypothèse d'induction, il existe une dérivation de la clause vide à partir de $S_1; A$. Or par la règle $\to_{n_g}$ nous obtenons l'ensemble de clauses $S_1; A; (A \to B)^\perp$ à partir de $S_1; (A \to B)^\perp$ et nous concluons grâce au lemme 5.3.4.

– $\dfrac{\Gamma \vdash B, \Delta}{\Gamma \vdash A \to B, \Delta} \to_{d_d}$: Par hypothèse d'induction, il existe une dérivation de la clause vide à partir de $T = S_1; B^\perp$. Or par la règle $\to_{n_d}$ nous obtenons l'ensemble de clauses $S_1; B^\perp; (A \to B)^\perp$ à partir de $S_1; (A \to B)^\perp$ et nous concluons grâce au lemme 5.3.4.

– $\dfrac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta} \neg_g$: Par hypothèse d'induction, il existe une dérivation de la clause vide à partir de $T = S_1; A^\perp$. Or par la règle $\neg_p$ nous obtenons l'ensemble de clauses $S_1; A^\perp; \neg A$ à partir de $S_1; \neg A$ et nous concluons grâce au lemme 5.3.4.

– $\dfrac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta} \neg_d$: Par hypothèse d'induction, il existe une dérivation de la clause vide à partir de $T = S_1; A$. Or par la règle $\neg_n$ nous obtenons l'ensemble de clauses $S_1; A; (\neg A)^\perp$ à partir de $S_1; (\neg A)^\perp$ et nous concluons grâce au lemme 5.3.4.

– $\dfrac{\Gamma, A[y/x] \vdash \Delta}{\Gamma, \exists x A \vdash \Delta} \exists_g$: Par hypothèse d'induction, il existe une dérivation de la clause vide à partir de $T = S_1; A(y)$. Or par la règle $\exists_p$ nous obtenons l'ensemble de clauses $S_1; A(y); \exists x.A(x)$ à partir de $S_1; \exists x.A(x)$ et nous concluons grâce au lemme 5.3.4. Remarquons que la condition sur la règle $\exists_g$ implique que la condition sur la règle $\exists_p$ est respectée.

– $\dfrac{\Gamma \vdash A[t/x], \Delta}{\Gamma \vdash \exists x A, \Delta} \exists_d$: Par hypothèse d'induction, il existe une dérivation de la clause vide à partir de $T = S_1; A(t)^\perp$. Or par la règle $\exists_n$ nous obtenons l'ensemble de clauses $S_1; A(t)^\perp; (\exists x.A(x))^\perp$ à partir de $S_1; (\exists x.A(x))^\perp$ et nous concluons grâce au lemme 5.3.4.

– $\dfrac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta} w_g$: l'induction et le lemme 5.3.4 permettent ici de conclure.

– $\dfrac{\Gamma \vdash \Delta}{\Gamma \vdash A, \Delta} w_d$: idem.

– $\dfrac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta} c_g$: Comme les ensembles de clauses sont des ensembles, nous concluons grâce à l'induction. Si l'on ne veut pas voir les choses ainsi, il suffit juste de considérer que l'on peut utiliser une clause à volonté dans le système, l'ensemble de clauses étant croissant.

– $\dfrac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta} c_d$: idem. □

V.3 Proposition 5.7.8

- $R = \vee_p$: S contient une clause $\Gamma, A \vee B$ et $C = \Gamma, A, B$. Nous devons donc prouver $\forall x_i(\Pi \vee (A \vee B)) \vdash_{LK} \forall y_i(\Pi \vee A \vee B)$.
  Les deux formules étant égales, c'est évident.
- $R = \vee_{n_g}$: S contient une clause $\Gamma, (A \vee B)^\perp$ et $C = \Gamma, A^\perp$. Nous devons donc prouver $\forall x_i(\Pi \vee \neg(A \vee B)) \vdash_{LK} \forall y_j(\Pi \vee \neg A)$.
  Si la première formule est valide dans une structure, prouvons que la seconde est valide. Si elle ne l'est pas, cela signifie qu'il y a une substitution σ des variables pour laquelle $(\Pi \vee \neg A)\sigma$ est non valide, ce qui signifie que $\Pi\sigma$ est non valide et $A\sigma$ est valide. Or si $\Pi\sigma$ est non valide, alors par hypothèse $\neg(A \vee B)\sigma$ est valide, ce qui signifie que $(A \vee B)\sigma$ est non valide, donc que $A\sigma$ est non valide, ce qui est contradictoire.
- $R = \vee_{n_d}$: S contient une clause $\Gamma, (A \vee B)^\perp$ et $C = \Gamma, B^\perp$. Nous devons donc prouver $\forall x_i(\Pi \vee \neg(A \vee B)) \vdash_{LK} \forall y_j(\Pi \vee \neg B)$.
  La preuve se fait de la même manière que précédemment, en remplaçant A par B.
- $R = \rightarrow_p$: S contient une clause $\Gamma, A \rightarrow B$ et $C = \Gamma, A^\perp, B$. Nous devons donc prouver $\forall x_i(\Pi \vee (A \rightarrow B)) \vdash_{LK} \forall y_j(\Pi \vee \neg A \vee B)$.
  Comme $A \rightarrow B$ est équivalent à $\neg A \vee B$ dans toute structure, c'est évident.
- $R = \rightarrow_{n_g}$: S contient une clause $\Gamma, (A \rightarrow B)^\perp$ et $C = \Gamma, A$. Nous devons donc prouver $\forall x_i(\Pi \vee \neg(A \rightarrow B)) \vdash_{LK} \forall y_j(\Pi \vee A)$.
  Si la première formule est valide dans une structure, prouvons que la seconde est valide. Si elle ne l'est pas, cela signifie qu'il y a une substitution σ des variables pour laquelle $(\Pi \vee A)\sigma$ est non valide, ce qui signifie que $\Pi\sigma$ est non valide et $A\sigma$ est non valide. Or si $\Pi\sigma$ est non valide, alors par hypothèse $\neg(A \rightarrow B)\sigma$ est valide, ce qui signifie que $(A \rightarrow B)\sigma$ est non valide, donc que $A\sigma$ est valide, ce qui est contradictoire.
- $R = \rightarrow_{n_d}$: S contient une clause $\Gamma, (A \rightarrow B)^\perp$ et $C = \Gamma, B^\perp$. Nous devons donc prouver $\forall x_i(\Pi \vee \neg(A \rightarrow B)) \vdash_{LK} \forall y_j(\Pi \vee \neg B)$.
  Si la première formule est valide dans une structure, prouvons que la seconde est valide. Si elle ne l'est pas, cela signifie qu'il y a une substitution σ des variables pour laquelle $(\Pi \vee \neg B)\sigma$ est non valide, ce qui signifie que $\Pi\sigma$ est non valide et $B\sigma$ est valide. Or si $\Pi\sigma$ est non valide, alors par hypothèse $\neg(A \rightarrow B)\sigma$ est valide, ce qui signifie que $(A \rightarrow B)\sigma$ est non valide, donc que $B\sigma$ est non valide, ce qui est contradictoire.
- $R = \wedge_n$: S contient une clause $\Gamma, (A \wedge B)^\perp$ et $C = \Gamma, A^\perp, B^\perp$. Nous devons donc prouver $\forall x_i(\Pi \vee \neg(A \wedge B)) \vdash_{LK} \forall y_j(\Pi \vee \neg A \vee \neg B)$.
  Comme $\neg A \wedge B$ est équivalent à $\neg A \vee \neg B$ dans toute structure c'est évident.
- $R = \wedge_{p_g}$: S contient une clause $\Gamma, A \wedge B$ et $C = \Gamma, A$. Nous devons donc prouver $\forall x_i(\Pi \vee (A \wedge B)) \vdash_{LK} \forall y_j(\Pi \vee A)$.
  Si la première formule est valide dans une structure, prouvons que la seconde est valide. Si elle ne l'est pas, cela signifie qu'il y a une substitution σ des variables pour laquelle $(\Pi \vee A)\sigma$ est non valide, ce qui signifie que $\Pi\sigma$ est non valide et $A\sigma$ est non valide. Or si $\Pi\sigma$ est non valide, alors par hypothèse $(A \wedge B)\sigma$ est valide, ce qui signifie que $(A \wedge B)\sigma$ est valide, donc que $A\sigma$ est valide, ce qui est contradictoire.
- $R = \wedge_{p_d}$: S contient une clause $\Gamma, A \wedge B$ et $C = \Gamma, B$. Nous devons donc prouver $\forall x_i(\Pi \vee (A \wedge B)) \vdash_{LK} \forall y_j(\Pi \vee B)$.
  La preuve est la même que précédemment en remplaçant A par B.
- $R = \neg_n$: S contient une clause $\Gamma, (\neg A)^\perp$ et $C = \Gamma, A$. Nous devons donc prouver $\forall x_i(\Pi \vee (\neg\neg A) \vdash_{LK} \forall y_j(\Pi \vee A)$.
  Comme $\neg\neg A$ est équivalent à A dans toute structure c'est évident.
- $R = \neg_p$: S contient une clause $\Gamma, \neg A$ et $C = \Gamma, A^\perp$. Nous devons donc prouver $\forall x_i(\Pi \vee \neg A) \vdash_{LK} \forall y_i(\Pi \vee \neg A)$.
  Les deux formules étant les mêmes c'est évident.
- $R = \exists_p$: S contient une clause $\Gamma, \exists x.A(x, x_1, \ldots, x_n)$ et
  $C = \Gamma, A(f(y_1, \ldots, y_n), y_1, \ldots, y_n)$ avec f une nouvelle fonction et y_k les variables libres de $A(y, y_1, \ldots, y_n)$ en dehors de y. Nous devons donc prouver

$\forall x_i(\Pi \vee \exists x.A(x, x_1, \ldots, x_n)) \vdash_{LK} \forall y_j(\Pi \vee A(f(y_1, \ldots, y_n), y_1, \ldots, y_n))$.
Dans ce cas c'est l'utilisation du lemme 5.7.4 qui permet de conclure de la même manière que dans le cas $\forall_n$.

- $R = \exists_n : S$ contient une clause $\Gamma, (\exists x.A(x))^\perp$ et $C = \Gamma, A(z)^\perp$ avec z une variable fraîche. Nous devons donc prouver $\forall x_i(\Pi \vee \neg\exists x.A(x)) \vdash_{LK} \forall y_j, z(\Pi \vee \neg A(z))$.
Si la première formule est valide dans une structure, prouvons que la seconde est valide. Si elle ne l'est pas, cela signifie qu'il y a une substitution σ des variables pour laquelle $(\Pi \vee A(z))\sigma$ est non valide, ce qui signifie que $\Pi\sigma$ est non valide et $A(z)\sigma$ est valide. Or si $\Pi\sigma$ est non valide, alors par hypothèse $\neg\exists x.A(x)\sigma$ est valide, ce qui signifie que $\exists x.A(x)\sigma$ est non valide, donc $A(z)\sigma$ en particulier est non valide, ce qui est contradictoire.
- $R = contr_p : S$ contient une clause Γ, A, A' et $C = \Gamma\sigma, A\sigma$ avec σ l'unificateur le plus général de A et A'. Nous devons donc prouver $\forall x_i(\Pi \vee A \vee A') \vdash_{LK} \forall y_j(\Pi\sigma \vee A\sigma)$.
Si la première formule est valide dans une structure, prouvons que la seconde est valide. Si elle ne l'est pas, cela signifie qu'il y a une substitution τ des variables pour laquelle $(\Pi\sigma \vee A\sigma)\tau$ est non valide, ce qui signifie que $\Pi\sigma\tau$ est non valide et $A\sigma\tau$ est non valide. Or si $\Pi\sigma\tau$ est non valide, alors par hypothèse $A\sigma\tau \vee A'\sigma\tau$ est valide, et comme $A\sigma = A'\sigma$ $A\sigma\tau$ est valide, ce qui est contradictoire.
- $R = contr_n : S$ contient une clause $\Gamma, A^\perp, A'^\perp$ et $C = \Gamma\sigma, A\sigma^\perp$ avec σ l'unificateur le plus général de A et A'. Nous devons donc prouver
$\forall x_i(\Pi \vee \neg A \vee \neg A') \vdash_{LK} \forall y_j(\Pi\sigma \vee \neg A\sigma)$.
C'est la même preuve que pour le cas précédent en remplaçant A et A' par leur négation.
- $R = res : S$ contient une clause Γ, A et une clause $\Gamma', A'^\perp$ et $C = \Gamma\sigma, \Gamma'\sigma$ avec σ l'unificateur le plus général de A et A'. Nous devons donc prouver
$\forall x_i(\Pi \vee A), \forall x'_j(\Pi' \vee \neg A') \vdash_{LK} \forall y_k(\Pi\sigma \vee \Pi'\sigma)$.
Si les deux formules à gauche du séquent sont valides dans une structure, prouvons que la formule de droite l'est également. Si elle ne l'est pas alors il y a une substitution τ des variables pour laquelle $(\Pi\sigma \vee \Pi'\sigma)\tau$ est non valide, ce qui signifie que $\Pi\sigma\tau$ et $\Pi'\sigma\tau$ sont non valides. Or par hypothèse, si elles sont non valides alors d'une part $\neg A'\sigma\tau$ est valide et d'autre part $A\sigma\tau$ est valide. Or $A'\sigma = A\sigma$, ce qui est contradictoire car on ne peut pas avoir une formule à la fois valide et non valide dans une structure. □

V.4 Proposition 5.7.12

- $F = A \vee B$: Posons $T = S'; C'_i, A, B$. Si $\phi(T)$ n'est pas contradictoire, alors il existe une structure dans laquelle toutes les formules de $\phi(T)$ sont valides. Donc les formules de $\phi(S')$ sont valides et $\phi(C'_i, A, B) = \forall x_i(C'_i \vee A \vee B)$ également. Or au moins une des $\phi(C_i) = \forall x_i(C' \vee A \vee B)$ n'est pas valide (car $\phi(S)$ est contradictoire) : contradiction.
- $F = (A \vee B)^\perp$: Posons $T = S'; C'_i, A^\perp; C'_i, B^\perp$. Si $\phi(T)$ n'est pas contradictoire alors il existe une structure dans laquelle toutes les formules de $\phi(T)$ sont valides. Donc les formules de $\phi(S')$ sont valides et $\phi(C'_i, A^\perp) = \forall x_i(C'_i \vee \neg A)$ et $\phi(C'_i, B^\perp) = \forall x_i(C'_i \vee (\neg B))$ également. Or au moins une des $\phi(C_i)$ n'est pas valide car $\phi(S)$ est contradictoire. Donc il existe une substitution σ telle que $C'_i\sigma \vee \neg(A \vee B)\sigma$ est non valide. Donc $C'_i\sigma$ est non valide et $\neg(A \vee B)\sigma$ est non valide. Or $\neg(A \vee B)$ est équivalent à $\neg A \wedge \neg B$ et donc soit $\phi(C'_i, A^\perp)$ soit $\phi(C'_i, B^\perp)$ n'est pas valide : contradiction.
- $F = A \to B$: Posons $T = S'; C'_i, A^\perp, B$. Si $\phi(T)$ n'est pas contradictoire, alors il existe une structure dans laquelle toutes les formules de $\phi(T)$ sont valides. Donc les formules de $\phi(S')$ sont valides et $\phi(C'_i, A^\perp, B) = \forall x_i(C'_i \vee \neg A \vee B$ également. Or au moins une des $\phi(C_i)$ n'est pas valide. Donc il existe une substitution σ telle que $C'_i\sigma \vee \neg(A \to B)\sigma$ est non valide. Donc $C'_i\sigma$ est non valide et $(A \to B)\sigma$ est non valide. Or $A \to B$ est équivalent à $\neg A \vee B$, donc $\phi(C'_i, A^\perp, B)$ ne peut être valide : contradiction.
- $F = (A \to B)^\perp$: Posons $T = S'; C'_i, A; C'_i, B^\perp$. Si $\phi(T)$ n'est pas contradictoire alors il existe une structure dans laquelle toutes les formules de $\phi(T)$ sont valides. Donc les formules de $\phi(S')$ sont valides et $\phi(C'_i, A) = \forall x_i(C'_i \vee A)$ et $\phi(C'_i, B^\perp) = \forall y_i(C'_i \vee \neg B)$ également.

Or une des formules $\phi(C_i)$ n'est pas valide. Donc il existe une substitution σ telle que $C'_i\sigma \vee \neg(A \to B)\sigma$ n'est pas valide. Donc $C'_i\sigma$ est non valide et $\neg(A \to B)\sigma$ est non valide. Or $\neg(A \to B)$ est équivalent à $A \wedge \neg B$ donc soit $\phi(C'_i, A)$ soit $\phi(C'_i, B^\perp)$ n'est pas valide : contradiction.

- $F = (A \wedge B)^\perp$: Posons $T = S'; C'_i, A^\perp, B^\perp$. Si $\phi(T)$ n'est pas contradictoire alors il existe une structure dans laquelle toutes les formules de $\phi(T)$ sont valides. Donc les formules de $\phi(S')$ sont valides et $\phi(C'_i, A^\perp, B^\perp) = \forall y_i(C'_i \vee \neg A \vee \neg B)$ également. Or au moins une des formules $\phi(C_i)$ n'est pas valide. Donc il existe une substitution σ telle que $C'_i\sigma \vee \neg(A \wedge B)\sigma$ n'est pas valide. Donc $C'_i\sigma$ est non valide et $\neg(A \wedge B)\sigma$ est non valide. Or $\neg(A \wedge B)$ est équivalent à $\neg A \vee \neg B$, donc $\phi(C'_i, A^\perp, B^\perp)$ n'est pas valide : contradiction.
- $F = A \wedge B$: Posons $T = S'; C'_i, A; C'_i, B$. Si $\phi(T)$ n'est pas contradictoire alors il existe une structure dans laquelle toutes les formules de $\phi(T)$ sont valides. Donc les formules de $\phi(S')$ sont valides et $\phi(C'_i, A) = \forall y_i(C'_i \vee A)$ et $\phi(C'_i, B) = \forall y_i(C'_i \vee B)$ également. Or au moins une des $\phi(C_i)$ n'est pas valide. Donc il existe une substitution σ telle que $C'_i\sigma \vee (A \wedge B)\sigma$ n'est pas valide. donc $C'_i\sigma$ est non valide et $(A \wedge B)\sigma$ est non valide. Donc $\phi(C'_i, A)$ ou $\phi(C'_i, B)$ n'est pas valide : contradiction.
- $F = (\neg A)^\perp$: Posons $T = S'; C'_i, A$. Si $\phi(T)$ n'est pas contradictoire, alors il existe une structure dans laquelle toutes les formules de $\phi(T)$ sont valides. Donc les formules de $\phi(S')$ sont valides et $\phi(C'_i, A) = \forall y_i(C'_i \vee A)$ également. Or au moins une des formules $\phi(C_i)$ n'est pas valide. Donc il existe une substitution σ telle que $C'_i\sigma \vee \neg\neg A\sigma$ n'est pas valide. Donc $C'_i\sigma$ est non valide et $\neg\neg A\sigma$ est non valide. Or $\neg\neg A$ est équivalent à A, donc $\phi(C'_i, A)$ n'est pas valide : contradiction.
- $F = \neg A$: Posons $T = S'; C'_i, A^\perp$. Si $\phi(T)$ n'est pas contradictoire, alors il existe une structure dans laquelle toutes les formules de $\phi(T)$ sont valides. Donc les formules de $\phi(S')$ sont valides et $\phi(C'_i, A^\perp) = \forall y_i(C'_i \vee \neg A)$ également. Or au moins une des formules $\phi(C_i) = \forall y_i(C'_i \vee \neg A)$ n'est pas valide : contradiction.
- $F = \exists x.A(x, x_1, \ldots, x_n)$: Posons $T = S'; C'_i, A(f(x_i), x_1, \ldots, x_n)$ avec f un nouveau symbole de fonctions, et x_i les variables libres de A en dehors de x. Nous savons par le lemme 5.7.4 que $\phi(T)$ est contradictoire si et seulement si $\phi(S)$ est contradictoire.
- $F = (\exists x.A(x))^\perp$: Posons $T = S'; C'_i, A(z)^\perp$ avec z une nouvelle variable. Si $\phi(T)$ n'est pas contradictoire, alors il existe une structure dans laquelle toutes les formules de $\phi(T)$ sont valides. Donc les formules de $\phi(S')$ sont valides et $\phi(C'_i, A(z)^\perp) = \forall y_i, z(C'_i \vee \neg A(z))$ également. Or au moins une des formules $\phi(C_i)$ n'est pas valide. Donc il existe une substitution σ telle que $C'_i\sigma \vee \neg\exists x.A(x)\sigma$ n'est pas valide. donc $C'_i\sigma$ est non valide et $\neg\exists x.A(x)\sigma$ est non valide. Or $\neg\exists x.A(x)\sigma$ est équivalent à $\forall x.\neg A(x)$. Ainsi en particulier $\neg A(z)\sigma$ est non valide donc $\phi(C'_i, A(z))$ n'est pas valide : contradiction. □

résumé

Cette Thèse est la conclusion de trois ans de travail sur un projet nommé DemoNat. Le but de ce projet est la conception d'un système d'analyse et de vérification de démonstrations mathématiques écrites en langue naturelle.
L'architecture générale du système se décrit en 4 phases :

1. analyse de la démonstration par des outils linguistiques ;
2. traduction de la démonstration dans un langage restreint ;
3. interprétation du texte traduit en un arbre de règles de déduction ;
4. validation des règles de déduction à l'aide d'un démonstrateur automatique.

Ce projet a mobilisé des équipes de linguistes et de logiciens, les deux premières phases étant la tâche des linguistes, et les deux dernières étant la tâche des logiciens.
Cette thèse présente plus en détail ce projet et développe principalement les points suivants :
– définition du langage restreint et de son interprétation ;
– propriétés du type principal de termes d'un λ-calcul typé avec deux flèches entrant dans le cadre d'un outil linguistique, les ACGs ;
– description du démonstrateur automatique.

abstract

This Thesis is the conclusion of three years of work in a project named DemoNat. The aim of this project is to design a system able to analyse and validate mathematical proofs written in a natural language.
The general scheme of the system is the following :

1. analysis of the proof by means of linguistics tools ;
2. translation of the proof in a restricted language ;
3. interpretation of the translated text in a deduction rules tree ;
4. validation of the deduction rules with an automatic prover.

This project envolved teams of linguists and logicians, the first two phases being the task of the linguists, and the lasts ones being the task of the logicians.
This thesis presents in more details the project and develops mainly the following points :
– Definition of the restricted language and its interpretation ;
– proprerties of the principal type of terms of a typed λ-calculus with two arrows, part of a linguistic tool, the ACGs ;
– Description of the automatic prover.